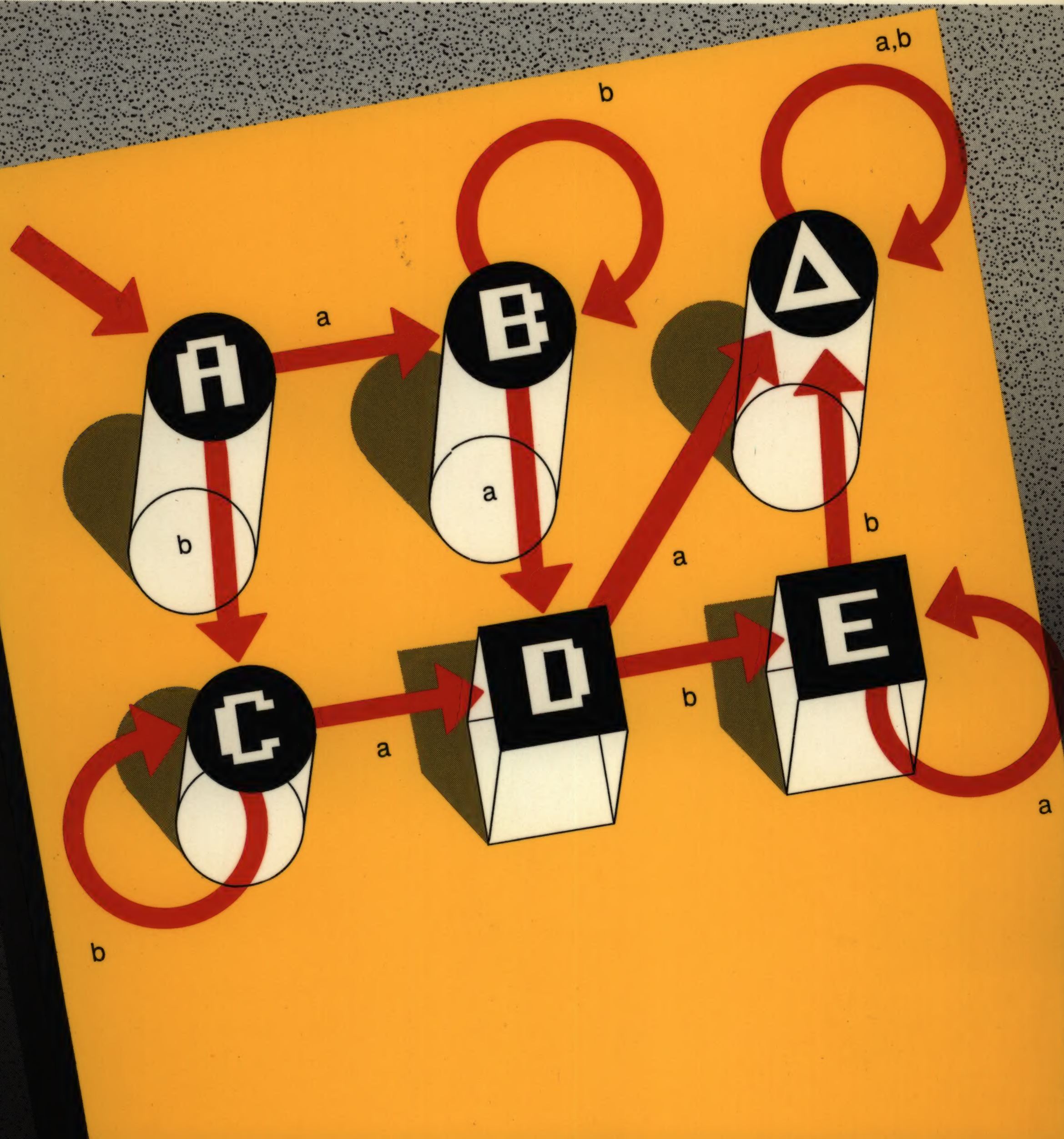
Inleiding in de theorie van formele talen

Dr. V.J. Rayward-Smith

ACADEMIC SERVICE



Inleiding in de theorie van formele talen

Serie onder redactie van **Ir. J.J. van Amstel**:

Database, een inleiding — Date
Inleiding systeemanalyse en systeemontwerp — Davis
Software Engineering — Sommerville
Problemen oplossen met de computer — Dromey
Compilerbouw — Wirth
Theorie en praktijk van besturingssystemen — Peterson & Silberschatz
Inleiding in de theorie van formele talen — Rayward-Smith
Inleiding in de berekenbaarheidstheorie — Rayward-Smith

Serie in samenwerking met Addison-Wesley:

Bestuurlijke informatieverzorging — Van Swigchem

Inleiding in de theorie van formele talen

Dr. V.J. Rayward-Smith

Vertaling van: *A first course in formal language theory*
Uitgegeven door Blackwell Scientific Publications Ltd.
© 1986

Vertaling: Drs. M.M. Stefanski

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Rayward-Smith, V.J.

Inleiding in de theorie van formele talen / V.J. Rayward-Smith ; [vert. uit het Engels door M.M. Stefanski]. - Schoonhoven : Academic service. - Ill.
Vert. van: A first course in formal language theory. - Oxford : Blackwell Scientific, 1986. - Met lit. opg.
ISBN 90-6233-242-0
SISO 803.5 SVS 8.12.3 UDC 800.89:800.92 NUGI 852
Trefw.: formele talen.

10 9 8 7 6 5 4 3 2 1

Uitgegeven door: Academic Service
Postbus 81
2870 AB Schoonhoven

Zetwerk: tedejo producties, Voorhout
Dit boek is zetklaar gemaakt met de programma-editor van WordPerfect Library, gezet met LaTeX (PCTeX) en afgedrukt op een QMS-PS 810 laserprinter.
Ontwerp omslag: BSE Advertising, Alphen a/d Rijn
Druk: Krips Repro Meppel
Bindwerk: Meeuwis Amsterdam

ISBN: 90 6233 242 0
NUGI: 852

To my family

Een boek zonder citaten is, me judice,
geen boek maar een stuk speelgoed.
–T.L. PEACOCK
Crotchet Castle

Bij de nederlandse vertaling

De meeste literatuur op het gebied van de theorie van formele talen is Engelstalig. Er is daarom nauwelijks sprake van een standaard Nederlandse terminologie voor de bij de behandeling van dit onderwerp gebruikte vaktermen. Gelukkig bestaat er in Nederland iemand die zowel expert is in de theorie van formele talen als in het gebruik van correct Nederlands. Ik doel op Prof. Dr. H. Brandt Corstius van wiens publicaties:

Algebraïsche taalkunde, Oosthoeks's Uitgeversmaatschappij, Utrecht 1974,
en
Computer-taalkunde, Coutinho, Muiderberg 1978,

ik dankbaar gebruik heb gemaakt.

Maarten Stefanski, Gouda, juni 1987

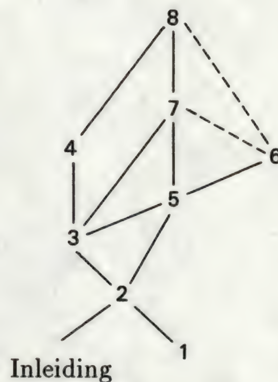
Inhoud

Voorwoord

Voor veel informaticastudenten is de theorie van formele talen een struikelblok omdat de notatie moeilijk te begrijpen is. Toch is het onderwerp niet te vermijden omdat de theorie van de compilerbouw op de formele taaltheorie is gebaseerd. Het boek is bedoeld voor eerste- en tweedejaars informaticastudenten en bevat voldoende ondergrond voor een cursus compilerbouw. In verband met dit laatste wordt vooral ingegaan op de theorie van de reguliere en contextvrije grammatica's.

De meeste boeken op dit gebied richten zich op universitair onderwijs in de tweede fase. De stof wordt dan diepgaander behandeld maar blijkt voor veel studenten nogal afschrikwekkend. In dit boek heb ik geprobeerd de onderwerpen zo eenvoudig mogelijk te presenteren, zonder daarbij een formele behandeling volledig te laten varen. In de eerste hoofdstukken behandel ik de bewijzen stap voor stap om duidelijk te maken hoe formele bewijzen worden geconstrueerd. Vrijwel steeds wordt van de inductieve bewijsmethode gebruik gemaakt en geleidelijk aan worden daarbij minder details vermeld. Tegen het einde van het boek worden in de meeste gevallen alleen schetsen van bewijzen gegeven en wordt de nadruk gelegd op praktische toepassing. In figuur 0.1 is aangegeven hoe de hoofdstukken onderling samenhangen.

Aan het einde van het tweede jaar in een driejarige informatica-opleiding behoort het merendeel van de stof uit dit boek te zijn bestudeerd. Een eenjarige cursus zou slechts de hoofdstukken 1, 2, 3 en 5 behoeven te dekken, met nog enkel onderdelen uit de hoofdstukken 4 en 6. Een afzonderlijk blok formele taaltheorie kan dit boek in circa eenentwintig college-uren behandelen.



Figuur 0.1

Tot het schrijven van dit boek werd ik vooral gemotiveerd door een groep studenten aan de universiteit van Californië in Santa Barbara, V.S. Tijdens mijn sabbatical-jaar onderwees ik daar formele taaltheorie aan jongerejaars studenten. Ik dank hen voor hun enthousiasme en voor de steun en aanmoediging die ik kreeg van de faculteitsafdeling en van mijn eigen universiteit tijdens de voorbereiding van dit boek. Met name Dr G.P. McKeown (East Anglia) wil ik danken voor zijn commentaar en Theodora Potter (Santa Barbara) voor haar uitstekende typewerk.

V.J. Rayward-Smith

Inleiding

Bij het bestuderen van een (al of niet formele) taal hebben we allemaal een bepaald voordeel. Iedereen is expert in minstens één taal: zijn of haar moedertaal. Of dit nu Engels, Frans, Spaans of Nederlands is, we zijn er zeer vaardig in en we ontwikkelen deze vaardigheid gedurende ons hele leven verder. Behalve één of meer natuurlijke talen kennen velen van ons tegenwoordig ook programmeertalen zoals Algol, Pascal, FORTRAN of BASIC. In deze talen worden programma's geschreven en zij moeten dus met computers kunnen communiceren. U zult wel, net als ik, gemerkt hebben dat computers niet erg bedreven zijn in het begrijpen van uw goedbedoelde programma's! In natuurlijke talen kunnen we heel goed met elkaar communiceren in slecht gestructureerde of onvolledige zinnen, maar computers hebben niet aan een half woord genoeg. Programma's moeten zich houden aan zeer strenge regels en de kleinste afwijking van die regels wordt als incorrect afgewezen.

In het ideale geval is een zin in een taal zowel *semantisch* (qua betekenis) als *syntactisch* (qua grammaticale structuur) correct. In het Nederlands of Engels zijn zinnen vaak syntactisch onjuist terwijl zij toch de bedoelde semantiek feilloos overdragen. In programmeertalen is syntactische correctheid essentieel, wil überhaupt een betekenis worden overgedragen.

Als we de semantiek van een zin bestuderen dan bestuderen we de betekenis van die zin. De volgende zinnen hebben allen dezelfde semantische interpretatie en dus dezelfde betekenis.

De man slaat de hond.
De hond wordt door de man geslagen.
L'homme frappe le chien.

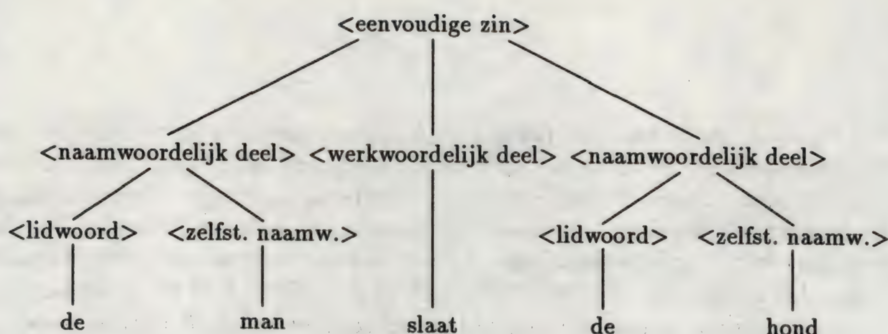
Het bestuderen van de syntaxis of de grammatica is het bestuderen van de zinsstructuur. De zin

De man slaat de hond.

kan worden ontleed in zijn samenstellende grammaticale delen.

De man	slaat	de hond
<naamwoordelijk deel>	<werkwoordelijk deel>	<naamwoordelijk deel>

Elke zin met deze structuur is syntactisch correct Nederlands. Een verzameling van deze eenvoudige zinnetjes kunnen we met behulp van de volgende regels beschrijven.



Figuur 0.2

<eenvoudige zin> ::= <naamwoordelijk deel><werkwoordelijk deel> <naamwoordelijk deel>
<naamwoordelijk deel> ::= <lidwoord><zelfstandig naamwoord>
<zelfstandig naamwoord> ::= auto
<zelfstandig naamwoord> ::= man
<zelfstandig naamwoord> ::= hond
<lidwoord> ::= de
<lidwoord> ::= een
<werkwoordelijk deel> ::= slaat
<werkwoordelijk deel> ::= eet

De regels zijn geformuleerd in BNF (Backus-Naur Form). Dit is een notatie die vaak wordt gebruikt voor het beschrijven van de syntaxis van programmeertalen. In hoofdstuk 2 gaan we hier verder op in.

In het voorbeeld wordt <eenvoudige zin> gedefinieerd als een <naamwoordelijk deel> gevolgd door een <werkwoordelijk deel> en opnieuw door een <naamwoordelijk deel>. Een <naamwoordelijk deel> bestaat uit een <lidwoord> gevolgd door een <zelfstandig naamwoord>. Kiezen we 'de' als lidwoord, 'man' als zelfstanding naamwoord, 'slaat' als werkwoordelijk deel, 'de' als tweede lidwoord en 'hond' als tweede zelfstandig naamwoord dan hebben we aangetoond dat 'de man slaat de hond' een syntactisch correcte <eenvoudige zin> is. Met behulp van een afleidingsboom, zoals is getekend in figuur 0.2, kan dit overzichtelijker worden gemaakt.

Totaal kan <eenvoudige zin> leiden tot $2 \times 3 \times 2 \times 2 \times 3 = 72$ verschillende zinnen. Zij zijn allemaal syntactisch correct, maar hun betekenis kan raadselachtig zijn. Wat betekent bijvoorbeeld 'de auto eet de man'?

Lees nu eens de volgende zin (geen <eenvoudige zin>):

Wij vliegen gieren naar Artis.

Een mogelijk ontleding is

Wij	vliegen	gieren	naar	Artis
<voornaam-woord>	<werk-woord>	<zelfstandig naamwoord>	<voor-zetsel>	<zelfstandig naamwoord>

De betekenis is een mededeling van enige piloten dat zij roofvogels naar een dierentuin te Amsterdam vliegen. Een andere ontleding is

Wij	vliegen	gieren	naar	Artis
<voornaam-woord>	<zelfstandig naamwoord>	<werkwoord>	<voor-zetsel>	<zelfstandig naamwoord>

een mededeling van een groep insecten met grote snelheid op weg naar dezelfde dierentuin.

Een andere ontleding leidt hier tot een geheel andere semantische interpretatie. Dit is wel een erg gekunsteld voorbeeld en in ons dagelijks taalgebruik komt dit soort misverstanden niet vaak voor, maar voor het werken met computers is dit fenomeen wel degelijk van belang. Een cruciale stap tijdens het compileren van een programma is het ontleden ervan om het semantisch te kunnen interpreteren, (de Engelse term hiervoor luidt 'parsing'). Syntaxisanalyse is daarom een belangrijk onderwerp binnen de informatica. Elke informaticus moet inzicht hebben in de structuur van grammatica's; een onderwerp dat trouwens ook voor de zuiver wiskundige interessant is.

Dit boek is weliswaar bedoeld voor de beginnende informaticus, maar de theorie van formele talen wordt vaak gezien als een onderdeel van de wiskunde en veel resultaten zijn zuiver mathematisch. Voor een diepgaandere behandeling beveel ik het boek van J.E. Hopcroft en J.D. Ullman aan: *Introduction to Automata theory, Languages and Computation*, Addison-Wesley, Reading, Mass.

Hoofdstuk 1

Basisbegrippen uit de Wiskunde

De wiskunde heeft niet alleen de waarheid in pacht, maar bezit ook grote schoonheid—zij is koel en streng als een beeldhouwwerk.

—LORD BERTRAND RUSSEL
The Study of Mathematics

In dit hoofdstuk wordt een overzicht gegeven van de wiskunde nodig om de rest van dit boek te kunnen begrijpen. Als deze stof nieuw voor u is, bestudeer hem dan nauwkeurig en probeer alle concepten volledig te begrijpen. Het boek *Mathematics for Computing* door G.P. McKeown en V.J. Rayward-Smith kan hierbij verdere ondersteuning geven. Als de stof niet nieuw voor u is dan is vluchtig doorlezen en eigen maken van de notatie voldoende.

1.1 Verzamelingen

Een verzameling is een collectie objecten zonder herhalingen. Elk object in een verzameling heet een element van die verzameling. Als het aantal elementen klein is dan kan een verzameling worden gespecificeerd door haar elementen op te sommen. Als D bijvoorbeeld de verzameling van dagen van de week is, dan is

$$D = \{\text{Maandag, Dinsdag, Woensdag, Donderdag, Vrijdag, Zaterdag, Zondag}\}.$$

We scheiden de elementen door komma's en omgeven ze door accolades. Meestal ligt tussen de elementen geen ordening vast, dus ook is

$$D = \{\text{Maandag, Woensdag, Vrijdag, Donderdag, Zondag, Dinsdag, Zaterdag}\}.$$

Als een element x in een verzameling A voorkomt dan schrijven we $x \in A$ (lees: x is element van A) en als x niet in A voorkomt dan schrijven we $x \notin A$ (lees: x is geen element van A).
Dus:

$$\text{Maandag} \in D$$

maar

Wasdag $\notin D$

Vaak heeft een verzameling zeer veel, misschien zelfs oneindig veel elementen. Opsommen wordt dan moeilijk of onmogelijk en we karakteriseren de verzameling door een *definiërende eigenschap*. Een element x zit in de verzameling als x die definiërende eigenschap bezit. Voor de verzameling D is zo'n eigenschap bijvoorbeeld:

'x is een dag van de week'.

We definiëren dan

$$D = \{x \mid x \text{ is een dag van de week}\}.$$

Dat wil zeggen D bestaat uit alle elementen x die deze definiërende eigenschap hebben. Een ander voorbeeld:

$$P = \{x \mid x \text{ is een priemgetal}\}$$

P is een oneindige verzameling van gehele getallen.

Als we verzamelingen op deze wijze definiëren dan moeten we ook het universum aangeven waaruit de elementen stammen. Als bijvoorbeeld

$$X = \{x \mid x > 2\}$$

dan moeten we ook weten welke waarden de elementen x mogen aannemen. Als het alleen om gehele getallen gaat dan is sprake van een andere verzameling dan wanneer x reële waarden mag aannemen. In het eerste geval geldt $2.1 \notin X$ in het tweede geval juist $2.1 \in X$. Als universum, $\mathcal{U}$, kan de verzameling der gehele getallen worden gekozen, maar ook die der reële getallen of bijvoorbeeld die der positieve gehele getallen, enzovoort.

Een bijzonder belangrijke verzameling is de *lege verzameling*. Deze bevat geen elementen en wordt door $\emptyset$, of door $\{\}$ weergegeven.

De verzameling A is een *deelverzameling* van de verzameling B, notatie $A \subset B$, als elk element van A ook in B zit. Als A geen deelverzameling van B is dan schrijven we $A \not\subset B$. Dus

$$1,2,4 \subset 1,2,3,4,5$$

maar

$$2,4,6 \not\subset 1,2,3,4,5.$$

Per definitie geld voor alle verzamelingen A

$$A \subset \mathcal{U}$$

en

$$\emptyset \subset A.$$

Twee verzamelingen A en B zijn gelijk, notatie $A = B$, als $A \subset B$ en $B \subset A$. Dus

$$1, 2, 3, 4 = 2, 1, 4, 3$$

maar

$$1, 2, 3, 4 \neq 2, 1, 3, 5.$$

Duidelijk is dat $A = B$ desd[1] als A en B precies dezelfde elementen bevatten.

Elementaire operaties op verzamelingen zijn de unaire operatie complement ($'$) en de binaire operaties vereniging ($\cup$), doorsnede ($\cap$) en verschil ($\backslash$). (Een unaire operatie werkt op één operand, een binaire heeft er twee nodig.) Deze operaties worden als volgt gedefinieerd. Als A en B verzamelingen zijn dan is

$$\begin{aligned} A' &= \{x \mid x \notin A\} \text{ alle elementen uit het universum maar niet in } A \\ A \cup B &= \{x \mid x \in A \text{ of } x \in B\} \text{ alle elementen in } A \text{ of in } B \\ A \cap B &= \{x \mid x \in A \text{ en } x \in B\} \text{ alle elementen in } A \text{ en in } B \\ A \backslash B &= \{x \mid x \in A, x \notin B\} \text{ alle elementen in } A\text{, maar niet in } B. \end{aligned}$$

Als bijvoorbeeld $\mathcal{U} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $A = \{0, 1, 3, 5\}$ en $B = \{2, 3, 5\}$ dan is

$$\begin{aligned} A' &= \{2, 4, 6, 7, 8, 9\}, \\ A \cup B &= \{0, 1, 2, 3, 5\}, \\ A \cap B &= \{3, 5\}, \\ A \backslash B &= \{0, 1\}, \\ B \backslash A &= \{2\}. \end{aligned}$$

Doorsnede en vereniging zijn beide *associatieve* operaties, dat wil zeggen voor alle verzamelingen A, B en C geldt

$$(A \cup B) \cup C = A \cup (B \cup C)$$

$$(A \cap B) \cap C = A \cap (B \cap C).$$

De verschiloperatie is niet associatief. Doorsnede en vereniging zijn ook *commutatief*. Dat wil zeggen

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A \text{ voor alle verzamelingen } A \text{ en } B.$$

De verschiloperator is niet commutatief.

Stelling 1.1 bevat een overzicht van een aantal eigenschappen van verzamelingen.

---

[1] Afkorting van 'dan en slechts dan'

Stelling 1.1. Eigenschappen van verzamelingen

Voor alle verzamelingen A, B en C in een universum $\mathcal{U}$ gelden:

1. De associatieve wetten
$$(A \cup B) \cup C = A \cup (B \cup C)$$
$$(A \cap B) \cap C = A \cap (B \cap C);$$

2. De commutatieve wetten
$$A \cup B = B \cup A$$
$$A \cap B = B \cap A;$$

3. De wetten van complementariteit
$$A \cup A' = \mathcal{U}$$
$$A \cap A' = \emptyset;$$

4. De wetten van gelijkmachtigheid
$$A \cup A' = A$$
$$A \cap A' = A;$$

5. De identiteitswetten
$$A \cup \emptyset = A$$
$$A \cap \mathcal{U} = A;$$

6. Wetten voor universum en lege verzameling
$$A \cup \mathcal{U} = \mathcal{U}$$
$$A \cap \emptyset = \emptyset;$$

7. De involutiewet
$$(A')' = A;$$

8. De wetten van de Morgan
$$(A \cup B)' = A' \cap B'$$
$$(A \cap B)' = A' \cup B';$$

9. De distributiviteitswetten
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C);$$

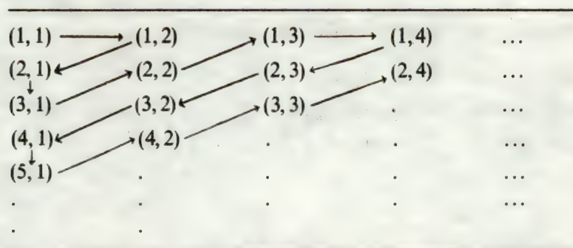
Omdat vereniging associatief is kunnen we ook schrijven $A \cup B \cup C$; de volgorde waarin de operaties worden toegepast heeft geen invloed op het resultaat. Dit kan worden uitgebreid naar $A_1 \cup A_2 \cup \ldots \cup A_n$ of

$$\bigcup_{i=1}^{n} A_i.$$

Evenzo schrijven we

$$\bigcap_{i=1}^{n} A_i$$

voor $A_1 \cap A_2 \cap \ldots \cap A_n$. Twee verzamelingen A en B heten disjunct als zij geen element gemeen hebben. Dit betekent dat $A \cap B = \emptyset$.



Tabel 1.1

1.2 Kardinaliteit en aftelbaarheid

Als A uit een eindig aantal elementen bestaat dan heet A een *eindige verzameling*. Is het aantal niet eindig dan heet A een *oneindige verzameling*.

Het kardinaalgetal $\#(A)$ van een eindige verzameling A is het aantal elementen in A. Als bijvoorbeeld D de verzameling van dagen van de week voorstelt dan is $\#(D) = 7$. Een verzameling met kardinaalgetal 1 heet een *singleton*.

Bij oneindige verzamelingen zou het prettig zijn als we een systematische lijst van de elementen konden maken, dus als we een eerste element, een tweede, een derde enz. konden aanwijzen. Als N bijvoorbeeld de verzameling der natuurlijke getallen voorstelt, dan ligt als opsomming de lijst $1, 2, 3, 4, \ldots$ voor de hand. We kunnen dan zonder dat dit tot misverstanden kan leiden spreken over het i^{de} natuurlijke getal. Het spreekt niet vanzelf (en het is zelfs niet waar) dat van elke oneindige verzameling zo'n lijst kan worden gemaakt.

Kunnen we bijvoorbeeld de elementen van Z, de verzameling van alle gehele getallen net zo opsommen als die van N? Het antwoord is: Ja, dat kan, bijvoorbeeld door afwisselend een positieve en een negatieve waarde op te schrijven:

$$0, +1, -1, +2, -2, +3, -3, \ldots$$

Elke $z \in Z$ komt ergens in deze lijst voor.

Een paar natuurlijke getallen schrijven we als (n_1, n_2) met $n_1, n_2 \in N$. De verzameling van deze paren is het produkt $N \times N$. Deze verzameling is in tabel 1.1 in tabelvorm weergegeven.

Steeds als we een manier kunnen vinden om de elementen van een verzameling systematisch op te sommen, kunnen we spreken van het i^{de} element uit die verzameling. De verzameling heet dan *aftelbaar*. Elke eindige verzameling is aftelbaar, maar dit geldt niet voor alle oneindige verzamelingen. Het halfopen interval $[0, 1)$, de verzameling van alle reële getallen ≥ 0 en < 1 is bijvoorbeeld niet aftelbaar. Dit kan worden bewezen met behulp van de *diagonalistatiemethode van Cantor*. We veronderstellen dat er een systematische opsomming bestaat van alle reële getallen uit het interval en we bewijzen dat deze veronderstelling tot een logische tegenspraak leidt.

Als we de getallen in de gebruikelijke decimale notatie weergeven dan bestaat

de volgende lijst als onze veronderstelling juist is.

1-ste reële getal	$0.a_{11}a_{12}a_{13}\ldots$
2-de reële getal	$0.a_{21}a_{22}a_{23}\ldots$
3-de reële getal	$0.a_{31}a_{32}a_{33}\ldots$
	$\ldots$
	$\ldots$

De elementen a_{ij} stellen één van de cijfers $0, 1, 2, \ldots, 9$ voor. Elk reëel getal uit $\in [0, 1)$ zou in deze lijst moeten voorkomen. Neem nu het getal

$$0.a_{11}a_{22}a_{33}\ldots$$

dat bestaat uit de decimalen op de hoofddiagonaal en construeer hieruit $0.b_{11}$ $b_{22}b_{33}\ldots$, met

$$b_{ii} = \begin{cases} a_{ii} + 1 & \text{als } a_{ii} < 9, \\ 0 & \text{als } a_{ii} = 9. \end{cases}$$

Het getal $0.b_{11}b_{22}b_{33}\ldots$ zit in $[0.1)$ maar komt niet voor in de oorspronkelijke lijst omdat het van het i^{de} getal in ieder geval in de i^{de} decimaal verschilt. De veronderstelling leidt dus tot een tegenspraak; $[0, 1)$ is niet aftelbaar.

1.3 Produkten van verzamelingen

A_1 en A_2 zijn verzamelingen. Het produkt $A_1 \times A_2$ bestaat uit de paren (a_1, a_2) met $a_1 \in A_1$ en $a_2 \in A_2$. Dus

$$A_1 \times A_2 = \{(a_1, a_2) \,|\, a_1 \in A_1, a_2 \in A_2\}.$$

Een eenvoudig voorbeeld. Als $A_1 = \{0, 1\}$ en $A_2 = \{x, y, z\}$ dan is $A_1 \times A_2 = \{(0, x), (0, y), (0, z), (1, x), (1, y), (1, z)\}$ Het produkt $N \times N$ kwam al bij de behandeling van aftelbaarheid te pas.

We breiden de definitie uit tot $A_1 \times A_2 \times A_3$, de verzameling van alle tripels (a_1, a_2, a_3) met $a_1 \in A_1, a_2 \in A_2, a_3 \in A_3$ en tot $A_1 \times A_2 \times \ldots \times A_n$ de verzameling van alle n-tupels $(a_1, a_2, \ldots, a_n)$ met $a_i \in A_i$ voor $i = 1, 2, \ldots, n$.

Voor eindige verzamelingen kan men aantonen dat geldt $\#(A_1 \times A_2) = \#(A_1) \times \#(A_2)$. Met behulp van een tabel kan men bewijzen dat $A_1 \times A_2$ aftelbaar is desd als A_1 en A_2 aftelbaar zijn.

Met behulp van inductie bewijst men dat

$$\#(A_1 \times A_2 \times \ldots \times A_n) = \#(A_1) \times \#(A_2) \times \ldots \times \#(A_n)$$

voor alle $n > 1$ als de verzamelingen $A_1, A_2, \ldots, A_n$ eindig zijn. Verder dat $A_1 \times A_2 \times \ldots \times A_n$ aftelbaar is desd als $A_1, A_2, \ldots A_n$ aftelbaar zijn.

Een *relatie* R is een willekeurige deelverzameling van $A1 \times A2$. $A1$ heet dan het *domein* van R en $A2$ heet het *bereik* of de *range* van R. We zullen vaak met

relaties te maken krijgen waarvan domein en range beide dezelfde verzameling A zijn. We noemen in dat geval $R \subset A \times A$ een relatie op A. Als bijvoorbeeld $A = 0, 1, 2, 3$ dan komt de verzameling geordende paren

$$L = \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$$

overeen met de relatie 'kleiner dan' en

$$E = \{(0,0),(1,1),(2,2),(3,3)\}$$

is de relatie 'is gelijk aan'. De gebruikelijke notatie voor $(a,b) \in R$ is aRb en voor $(a,b) \notin R$ is $a\not{R}b$.

Een relatie R op A heet *reflexief* als

$$aRa \text{ voor alle } a \in A.$$

De bovengenoemde relatie L is dus niet reflexief op A omdat immers $(0,0) \notin L$. De relatie E is wel reflexief.

Een relatie heet *symmetrisch* als

$$aRb \text{ impliceert dat } bRa.$$

L is niet symmetrisch omdat $(0,1) \in L$, maar $(1,0) \notin L$. E is wel symmetrisch.

R heet *transitief* op A als

$$aRb \text{ en } bRc \text{ impliceert dat } aRc.$$

L en E zijn beide transitief.

Als een relatie R op A reflexief, symmetrisch en transitief is dan heet de relatie een *equivalentierelatie*. De relatie E is een equivalentierelatie op $A = 0, 1, 2, 3$, maar dit geldt bijvoorbeeld ook voor de volgende relatie

$$V = \{(0,0),(0,2),(1,1),(1,3),(2,2),(2,0),(3,1),(3,3)\}.$$

Als R een equivalentierelatie op A is en $a \in A$ dan kan men een verzameling elementen $\overline{a}$ aangeven die via R met a een relatie hebben. Dus

$$\overline{a} = \{b \in A \mid aRb\}.$$

Een dergelijke verzameling noemt men een *equivalentieklasse*.

Bij E bestaan er vier verschillende equivalentieklassen, namelijk: $\overline{0} = \{0\}$, $\overline{1} = \{1\}$, $\overline{2} = \{2\}$ en $\overline{3} = \{3\}$. Bij V horen twee equivalentieklassen: $\overline{0} = \overline{2} = \{0, 2\}$ en $\overline{1} = \overline{3} = \{1, 3\}$ We kunnen de volgende stelling bewijzen.

Stelling 1.2

De equivalentieklassen bij een equivalentierelatie R op een verzameling A verdelen A in een aantal disjuncte, niet-lege verzamelingen.
Bewijs: Elke equivalentieklasse $\overline{a}$ is niet-leeg omdat immers $a \in \overline{a}$, (aRa want R is reflexief). Verder moeten we aantonen dat ofwel geldt $\overline{a} = \overline{b}$, ofwel $\overline{a} \cap \overline{b} = \emptyset$.

Als $\overline{a} \cap \overline{b} \neq \emptyset$ dan bestaat er een $c \in \overline{a} \cap \overline{b}$. Dus dan geldt $c \in \overline{a}$ en $c \in \overline{b}$ en dus aRc en bRc. Omdat R ook symmetrisch is volgt uit bRc dat cRb en omdat R transitief is volgt uit aRc en cRb dat aRb en dus is $b \in \overline{a}$. Voor alle $x \in \overline{b}$ geldt bRx en omdat aRb en R transitief is, geldt ook aRx en is dus $x \in \overline{a}$. Dus geldt $\overline{b} \subset \overline{a}$. Op overeenkomstige wijze kan worden aangetoond dat $\overline{a} \subset \overline{b}$ en dus geldt $\overline{a} = \overline{b}$, q.e.d.

We geven nog een voorbeeld van equivalentieklassen. Zij R de relatie op de positieve gehele getallen N, waarvoor geldt aRb desd als $|a - b|$ deelbaar is door 5. Er bestaan dan vijf verschillende equivalentieklassen met ieder een oneindig aantal elementen.

$$\begin{aligned} \overline{1} = \overline{6} = \overline{11} = \dots, \\ \overline{2} = \overline{7} = \overline{12} = \dots, \\ \overline{3} = \overline{8} = \overline{13} = \dots, \\ \overline{4} = \overline{9} = \overline{14} = \dots, \\ \text{en } \overline{5} = \overline{10} = \overline{15} = \dots, \end{aligned}$$

Als R een willekeurige relatie op A is dan heet de kleinste reflexieve (of symmetrische of transitieve) relatie op A met R als deelverzameling de reflexieve (symmetrische of transitieve) afsluiting van R. Als bijvoorbeeld

$$R = \{(0,1), (1,1), (1,2)\}$$

een relatie op $0, 1, 2$ is dan is de reflexieve afsluiting

$$\{(0,0), (0,1), (1,1), (1,2), (2,2)\},$$

de symmetrische afsluiting is

$$\{(0,1), (1,)), (1,1), (1,2), (2,1)\}$$

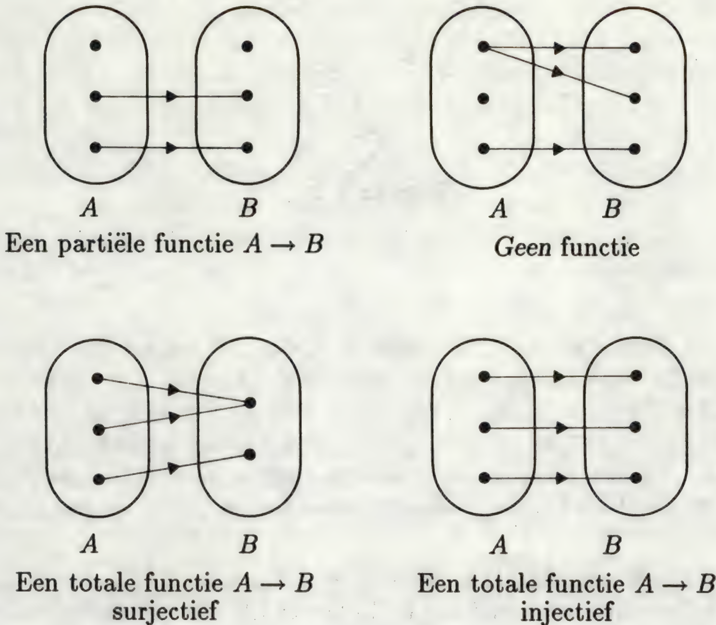
en de transitieve afsluiting is

$$\{(0,1), (0,2), (1,1), (1,2)\}.$$

Om nog een voorbeeld te geven: de relatie $<$, gedefinieerd op alle gehele getallen heeft als reflexieve afsluiting $\leq$ en als symmetrische afsluiting $\neq$. Omdat de relatie transitief is, is de transitieve afsluiting gelijk aan de relatie zelf.

Een functie $f : A \to B$ kan worden beschouwd als de verzameling van alle paren (a, b) met $a \in A$, $b \in B$ en $b = f(a)$. Een functie $f : A \to B$ kan dus worden gedefinieerd als een relatie in $A \times B$ met de eigenschap dat als $(a, b) \in f$ en $(a, c) \in f$ dat dan $b = c$. Dit betekent dat de waarde $f(a)$ uniek moet zijn. Als f niet op alle elementen van A is gedefinieerd dan noemen we f een *partiële functie* $A \to B$. Is f wel voor alle elementen van A gedefinieerd dan heet f een *totale functie* $A \to B$.

Een totale functie $f : A \to B$ heet *surjectief* als er voor elke $b \in B$ een $a \in A$ bestaat zo dat $f(a) = b$. De functie f heet *injectief* als unieke elementen uit A op unieke elementen uit B worden afgebeeld, dus als uit $f(a1) = f(a2)$ volgt dat $a1 = a2$. In figuur 1.1 hebben we enkele functies schematisch weergegeven.



Figuur1.1

Elke $(a, b) \in f$ wordt aangegeven door een lijnverbinding tussen $a \in A$ en $b \in B$.

Een totale functie $f : A \to B$ die zowel surjectief als injectief is, noemen we een *bijectie* of *een-eenduidige afbeelding*.

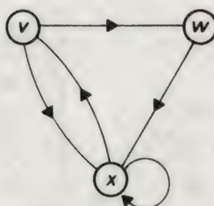
Aftelbaarheid kunnen we nu formeel als volgt definiëren: een verzameling A is aftelbaar als A of eindig is, of als er een bijectie $f : A \to N$ bestaat. Deze een-eenduidige afbeelding op de verzameling der natuurlijke getallen maakt een opsomming mogelijk, waarbij het i^{de} element van A wordt afgebeeld op het natuurlijke getal i.

1.4 Grafen en bomen

Een *gerichte graaf* of *digraaf*, $G = (V, E)$ bestaat uit een eindige verzameling *knopen* (*vertices*), V en een relatie E op V. Een digraaf kan als volgt in een plaatje worden weergegeven: voor elke $v \in V$ tekenen we een knoop v en als $(v, w) \in E$ dan verbinden we de knoop v met de knoop w door middel van een lijn. We krijgen dan



In figuur 1.2 is de digraaf $G_1 = (V_1, E_1)$ getekend met $V_1 = \{v, w, x\}$ en



Figuur 1.2

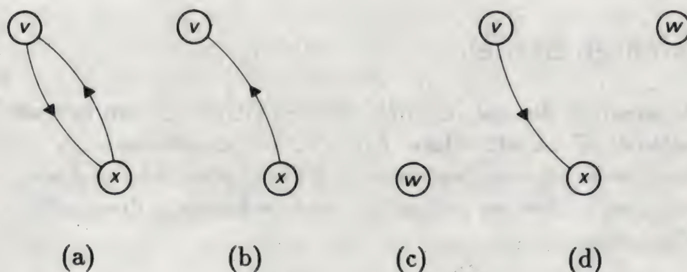
$E_1 = \{(v,w),(v,x),(w,x),(x,v),(x,x)\}$.

Als voor een digraaf $G = (V, E)$, $(v, w) \in E$ dan zeggen we dat v en w verbonden zijn. De verbinding wordt wel een 'kant', (Engels: edge) genoemd. Als er een rij knopen $v = v_o, v_1, \ldots, v_n = w$ met $n \geq 0$ bestaat zodanig dat $(v_i, v_{i+1}) \in E$ voor i=0,1,...,n-1 dan zegt men dat er een gericht pad ter lengte n bestaat in G van v naar w. Als er een pad van v naar w bestaat dan geldt ofwel $v = w$, ofwel we kunnen langs de verbindingen tussen de knopen vanuit v, w bereiken.

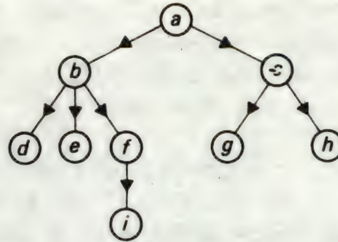
Als $G = (V, E)$ een digraaf is dan heet elke digraaf (V', E') met $V' \subset V$ en $E' \subset E$ een (gerichte) deelgraaf van G. In figuur 1.3 zijn vier deelgrafen van de digraaf G_1 uit figuur 1.2 getekend. Twee deelgrafen heten *disjunct* als zij geen knooppunten gemeen hebben. In figuur 1.3 zijn (a) en (c) disjunct, evenals (b) en (c).

Een belangrijk soort digraaf die we in de inleiding al tegenkwamen, is de boom. Een boom kan als volgt recursief worden gedefinieerd: een boom $T = (V, E)$ is een digraaf, waarin een bepaalde knoop r als wortel kan worden aangewezen. Als V uit slechts één knoop bestaat dan is $T = (\{r\}, \emptyset)$, anders bestaat T uit de wortel r en disjuncte deelgrafen $T_1, T_2, \ldots, T_k (k \geq 1)$ die weer bomen zijn te zamen met verbindingen vanuit r naar elke van deze bomen. In figuur 1.4 is een voorbeeld van een boom getekend. Ga na dat dit volgens onze definitie inderdaad een boom is.

We tekenen bomen gewoonlijk met de wortel bovenaan. Soms tekent men pijlen in de richting van de volgende knopen, maar deze zijn eigenlijk overbodig



Figuur 1.3



Figuur 1.4

en we zullen ze daarom in het vervolg weglaten.

Een knoop zonder uitgaande kanten heet een *externe knoop*, een *blaadje* of een *eindpunt*. In figuur 1.4 zijn de knopen d, e, i, g en h de blaadjes.

Een knoop die geen blaadje is heet een interne knoop. Veel van de terminologie in verband met bomen stamt uit de genealogie. Als er bijvoorbeeld een directe verbinding bestaat vanuit een knoop v naar een knoop w dan heet v de ouder van w. Alle knopen behalve de wortel hebben ouders. Als v de ouder van w is dan heet w het *kind* van v. Alle interne knopen hebben kinderen. Als er een gericht pad bestaat vanuit v naar w, langs een of meer kanten, dan heet v de *voorouder* van w en w de *nakomeling* van v.

1.5 Strings

Met een *string* bedoelen we een eindige rij symbolen $a_1a_2 \ldots a_n$. De elementen a_i zijn elementen van een of ander eindig alfabet Σ; in de string zijn herhalingen van hetzelfde teken toegestaan. Zo is 001110 een voorbeeld van een string met symbolen uit het alfabet $\Sigma = \{0, 1\}$.

Van een string met m symbolen zeggen we dat hij *lengte* m heeft; 001110 is dus een string met lengte 6. Een string zonder symbolen heeft lengte 0 en heet de *lege string* ε. De lengte van een string x wordt aangegeven door $|x|$. Dus $|001110| = 6$ en $|\varepsilon| = 0$.

Werken met strings is qua wiskunde vrij eenvoudig. Eerst moeten we een *alfabet* specificeren. Dit is een niet-lege eindige verzameling symbolen waarmee we strings kunnen opbouwen. Als ook het lege symbool of de spatie in onze strings moet kunnen voorkomen dan moet dit element van ons alfabet zijn. Ter wille van de duidelijkheid wordt de spatie door het teken $\wedge$ weergegeven. In plaats van 00 11 01 schrijven we dus $00 \wedge 11 \wedge 01$. De spatie $\wedge$ moet niet worden verward met de lege string; het is een string met lengte 1.

De verzameling van alle mogelijke eindige strings over een alfabet Σ zullen we noteren als Σ^*. Als $\Sigma = \{0, 1\}$ dan is

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, \ldots\}.$$

Σ^* heeft altijd een aftelbaar oneindig aantal elementen, (zie oefening 1.3). We gebruiken de normale notatie uit de verzamelingenleer $x \in \Sigma^*$ om aan te geven dat een string x element van Σ^* is en $\{x \notin \Sigma^*\}$ als dit niet het geval is. Op grond van de definitie van de lege string geldt $\varepsilon \in \Sigma^*$ voor alle verzamelingen Σ.

Als $x \in \Sigma^*$ een string ter lengte m is dan schrijven we $x = a_1a_2 \ldots a_m$ met $a_i \in \Sigma$ voor $1 \leq i \leq m$. Als $x \in \Sigma^*$ een string is met lengte m en $y \in \Sigma^*$ is een string met lengte n dan heet de string die wordt gevormd door de string y aan de string x toe te voegen de *concatenatie* van x en y, notatie xy. De lengte van xy is $m + n$. Als $x = a_1a_2 \ldots a_m$ en $y = b_1b_2 \ldots b_n$ dan is dus $xy = a_1a_2 \ldots a_mb_1b_2 \ldots b_n$. De operatie concatenatie is associatief: $(xy)z = x(yz)$, maar niet commutatief omdat meestal $xy \neq yx$. De lege string is een eenheidselement ten opzichte van concatenatie want $\varepsilon x = x\varepsilon = x$ voor alle $x \in \Sigma^*$.

Als een string $z \in \Sigma^*$ van de vorm xy is met $x, y \in \Sigma^*$ dan heet x een *prefix* van z en y heet een *postfix* van z. Als bijvoorbeeld $z = 00110$ dan is volgens de definitie ε een prefix van z, evenals $0, 00, 001, 0011$ en z zelf. De postfixen van z zijn $0, 10, 110, 0110$ en opnieuw z zelf.

Als $x, z \in \Sigma^*$ zodanig dat $z = wxy$ voor een $w, y \in \Sigma^*$ dan heet x een substring van z. De substrings van $z = 00110$ zijn ε, 0, 1, 00, 01, 10, 11, 001, 011, 110, 0011, 0110 en z zelf.

Een (formele) taal L over een alfabet Σ wordt eenvoudig gedefinieerd als elke deelverzameling van Σ^*. Als L_1 en L_2 twee van die talen zijn dan is hun (verzameling)concatenatie de taal $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$. Als bijvoorbeeld $L_1 = \{01, 0\}$ en $L2 = \{\varepsilon, 0, 10\}$ dan is $L_1L_2 = \{01, 0, 010, 00, 0110\}$. Net als stringconcatenatie is verzamelingconcatenatie associatief maar niet commutatief. Voor iedere taal L geldt dat $L\{\varepsilon\} = \{\varepsilon\}L = L$ en de singleton verzameling $\{\varepsilon\}$ is dus een eenheidselement voor verzamelingconcatenatie. Merk op dat de singleton met de lege string niet hetzelfde is als de lege verzameling $\emptyset$. Deze laatste is een nulelement voor verzamelingconcatenatie, want $L\emptyset = \emptyset L = \emptyset$ voor iedere taal L.

De concatenatie van een taal L met zichzelf is LL en wordt geschreven als L^2. De algemene definitie luidt $L^0 = \{\varepsilon\}$, $L^1 = L$, $L^i = LL^{i-1} = L^{i-1}L$ voor $i \geq 2$. De *Kleene-afsluiting* van L, notatie L^* is gedefinieerd als

$$\bigcup_{i=0}^{\infty} L^i.$$

L^+ is per definitie

$$\bigcup_{i=1}^{\infty} L^i.$$

en dus geldt $L^* = L^+ \cup \{\varepsilon\}$. Σ^2 is de notatie voor alle strings in Σ^* met lengte twee, Σ^3 voor die met lengte drie, etc. De verzameling van alle strings met lengte groter gelijk een wordt geschreven als Σ^+ en dus is

$$\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i$$

en

$$\Sigma^* = \Sigma^+ \cup \{\varepsilon\} = \bigcup_{i=0}^{\infty} \Sigma^i.$$

1.6 Oefeningen

1. Als $A = \{ab, c\}$ en $B = \{c, ca\}$ twee talen zijn in $\{a, b, c\}^*$ bepaal dan
   (a) $A \cup B$,
   (b) $A \backslash B$,
   (c) $A' \backslash B'$,
   (d) AB,
   (e) BA,
   (f) $A^2 \cup B^2$.
2. Bewijs dat de verzameling van de rationale getallen aftelbaar oneindig is.
3. Als Σ een eindig alfabet is dan is Σ^* aftelbaar oneindig. Bewijs dit.
4. De diepte van een knoop v in een boom $T = (V, E)$ wordt als volgt gedefinieerd: als v de wortel is dan is de diepte van v in T gelijk aan 0; als v niet de wortel is dan is $v \in T_i$ voor een of andere subboom T_i die via een enkele verbinding met de wortel van T is verbonden. De diepte van v in T is dan een meer dan de diepte van v in T_i.

   (a) Bepaal de dieptes van alle knopen in de boom in figuur 1.4. De diepte van de boom is de maximale diepte van een knooppunt in de boom.

   (b) Als T een boom met diepte k is en als elke knoop precies twee kinderen heeft dan heeft T n knopen, waarbij

   $$2k + 1 \leq n \leq 2^{k+1} - 1$$

   Bewijs dit.

5. Zij T een boom. Bewijs dat er een uniek pad bestaat vanuit de wortel naar elk knooppunt in de boom. (Hint: bewijs dit met inductie.)
6. Als $x \in \Sigma^*$ dan wordt de omkering van x, x^r als volgt recursief gedefinieerd. Als $x = \varepsilon$ dan is $x^r = \varepsilon$ anders $x = ay$ met $a \in \Sigma$ en $y \in \Sigma^*$ en dan geldt $x^r = y^r a$. Formuleer een recursieve functie voor $|x|$, de lengte van een string $x \in \Sigma^*$ en bewijs met inductie dat $|x| = |x^r|$ voor alle $x \in \Sigma^*$. (Hint: maak gebruik van het bewijs dat $|xy| = |x| + |y|$ voor alle $x, y \in \Sigma^*$.)
7. Als R_1 en R_2 twee relaties op A zijn dan geldt:
   (a) als R_1 reflexief is dan geldt dit ook voor $R_1 \cup R_2$;
   (b) als R_1 en R_2 reflexief zijn dan geldt dat ook voor $R_1 \cap R_2$.
   (c) Blijven deze beweringen geldig als we 'reflexief' vervangen door 'symmetrisch'? En als we 'transitief' substitueren?

8. Zij R een relatie in $A \times B$ en zij S een relatie in $B \times C$. De compositie van R met S, R_0S is gedefinieerd als de relatie in $A \times C$ met

$$R_0S = \{(a,c) | \text{ er bestaat een } b \in B \text{ zodanig dat: } (a,b) \in R \text{ en } (b,c) \in S\}$$

(a) Bewijs dat compositie een associatieve operatie is.
(b) Als R en S surjectieve totale functies zijn dan is R_0S een surjectieve totale functie van A op C.

9. Als $R \subset A \times B$ een relatie is dan is haar *inverse* R^{-1} een relatie in $B \times A$ gedefinieerd door

$$R^{-1} = \{(b,a) \,|\, (a,b) \in R\}.$$

(a) Bewijs dat $(R^{-1})^{-1} = R$.
(b) Welke voorwaarden moeten we aan R opleggen als R^{-1} een totale functie is?

Als $A = B$ en R dus een relatie op A is, bewijs dan dat

(c) R is reflexief desd als R^{-1} reflexief is;
(d) R is symmetrisch desd als R^{-1} symmetrisch is;
(e) R is transitief desd als R^{-1} transitief is.

10. De verzameling van alle deelverzamelingen van een verzameling A wordt aangegeven door 2^A en heet de *machtverzameling* van A.
(a) Als $A = a, b, c$ schrijf dan de acht elementen van 2^A uit.
(b) Bewijs dat $\#(2^A) = 2^n$ als $\#(A) = n$.

11. Een functie $f : 2^A x 2^A \rightarrow 2^A$ heet *monotoon stijgend* als uit $A_1 \subset B_1$ en $A_2 \subset B_2$ volgt dat $f(A_1, A_2) \subset f(B_1, B_2)$. Bewijs dat vereniging en verzamelingconcatenatie beide monotoon stijgende functies zijn.

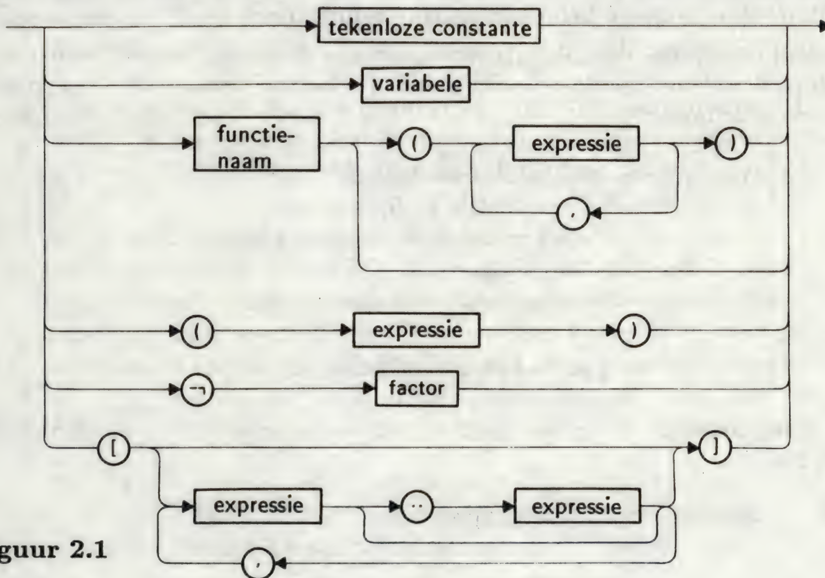
12. Zij $G = (V, E)$ een digraaf en zij E^* de reflexieve afsluiting van de transitieve afsluiting van E. Bewijs dat er een gericht pad bestaat van v naar w in G, met $v, w \in V$ dan en slechts dan als $(v, w) \in E^*$.

Hoofdstuk 2

Grammatica's; een inleiding

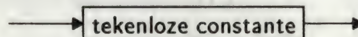
In het eindige openbaart zich het oneindige.
—THEODORE ROETHKE
The Far Field

In de meeste verhandelingen over hogere programmeertalen wordt aandacht besteed aan een formele definitie van de syntaxis. Pascal wordt bijvoorbeeld vaak beschreven met behulp van *syntaxisdiagrammen*. We gaan zo'n syntaxisdiagram eens nader bekijken. In figuur 2.1 is het begrip 'factor' gedefinieerd.



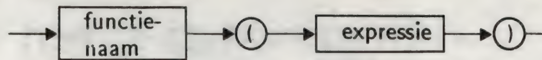
Figuur 2.1

Door het diagram kunnen verschillende routes worden afgelegd, die allen één of meer knooppunten bevatten. Bijvoorbeeld

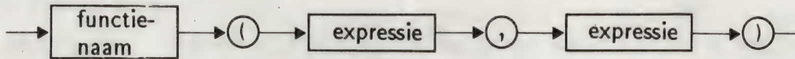


Dit is een route die maar één knooppunt bevat. De volgende iets ingewikkelder

route bevat vier knooppunten



Nog ingewikkelder is



Omdat het diagram lussen bevat is er in feite een oneindig aantal routes mogelijk en elke route levert een toegelaten definitie van het begrip 'factor'. Met syntaxisdiagrammen kunnen dit soort definities op een eenvoudige en bondige wijze worden gegeven. Als u probeert 'factor' alleen in woorden te definiëren zult u zien hoe moeizaam dat gaat.

Elk knooppunt in het diagram dat is omgeven door een rechthoekje, wordt gedefinieerd in een ander diagram. Als 'factor' dus wordt gedefinieerd als variabele kan vervolgens in het diagram voor 'variabele' de syntaxis van dit begrip worden opgezocht. De grootheden in de rechthoekjes heten *nonterminale* grootheden. Een omcirkeld knooppunt, zoals (is een *terminale* grootheid; het is een element van de door de grammatica gedefinieerde taal.

De syntaxis van Pascal kan ook worden gedefinieerd met behulp van de BNF-notatie die we in de inleiding bespraken. In BNF wordt een nonterminale grootheid omgeven door de symbolen < en >. Het teken ::= betekent 'is gedefinieerd als' en | wordt gebruikt voor 'of'. 'Factor' kan nu als volgt in BNF worden gedefinieerd

<factor> ::= <tekenloze constante>
| <variabele> | <functienaam>
| <functienaam> (<expressielijst>)
| (<expressie>)
| ¬ <factor>
| [ ]
| [<twee-expressielijst>]

De nonterminalen <expressielijst> en <twee-expressielijst> worden gedefinieerd als

<expressielijst> ::= <expressie>
| <expressie>,<expressielijst>

en

<twee-expressielijst> ::= <twee-expressie>
| <twee-expressie>,<twee-expressielijst>

met

<twee-expressie> ::= <expressie>
| <expressie> .. <expressie>

Bij het definiëren van programmeertalen kan men net zo goed syntaxisdiagrammen als BNF-notatie gebruiken; zij zijn volledig uitwisselbaar. Jensen en Wirth geven in hun *Pascal: User Manual and Report* (Springer) de syntaxisdefinitie op beide manieren. Vanuit het begrip <programma> kunnen via de syntaxisregels alle toegelaten programma's worden afgeleid. Toch zijn we er niet helemaal als we de syntaxis van de taal hebben beschreven. Aan het gebruik van Pascal bijvoorbeeld zijn verdere beperkingen opgelegd die niet in de syntaxisregels tot uitdrukking komen, zoals de regel dat elke variabele die we in een programma gebruiken in een declaratie moet worden genoemd. Een dergelijke regel is niet te vangen in BNF-notatie of in daarmee equivalente beschrijvingen. Eén van de belangrijke onderwerpen van de theorie van formele talen is dan ook het onderzoek naar de kracht en de beperkingen van BNF-achtige notaties.

2.1 Frasestructuurgramatica's

De formele grammaticadefinitie die we zullen gebruiken heeft overeenkomsten met de BNF-notatie. In ons formele model gebruiken we hoofdletters voor nonterminalen, kleine letters voor terminalen. Het teken ::= vervangen we door $\rightarrow$. Een eenvoudige grammatica ziet er dan zo uit

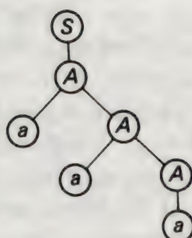
$$S \rightarrow A|B$$
$$A \rightarrow aA|a$$
$$B \rightarrow bB|b$$

Uit S kunnen we A of B afleiden. Uit A kunnen we in één stap de string a afleiden, in twee stappen de string aa en in k stappen de string a^k. Op dezelfde manier kunnen we uit B in k stappen de string b^k ($k \geq 1$) afleiden. Als we een string β uit een string α kunnen afleiden door het toepassen van één van de produktieregels uit de grammatica dan schrijven we $\alpha \Rightarrow \beta$. Als $\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{n-1} \Rightarrow \alpha_n$ ($n \geq 1$) dan korten we dit af tot $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ of zelfs tot $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$. Als we de hierboven vermelde grammatica gebruiken dan geldt

$$S \Rightarrow A \Rightarrow aA \Rightarrow aaA \Rightarrow aaa$$

Dit is een toegelaten afleiding van a^3 uit S. De afleiding kan ook door een boom worden voorgesteld, zoals in figuur 2.2.

We gaan deze begrippen nu verder formaliseren. In onderstaande definitie worden de afleidingsregels (die overeenkomen met het formele begrip produktieregel) gegeneraliseerd in die zin dat zowel in het linkerlid als in het rechterlid strings van terminalen of nonterminalen worden toegelaten. Tot nu toe gebruikten we steeds voorbeelden met precies één nonterminaal in het linkerlid en voor de bestudering van programmeertalen zijn dat ook de interessantste voorbeelden.



Figuur 2.2

Een *frasestructuurgrammatica* (Engels: phrase structure grammar, afgekort als PSG) is een 4-tupel (N, T, P, S). Hierbij is

(a) N een eindige verzameling van *nonterminalen*, weergegeven door hoofdletters (mogelijk met subscripts);
(b) T is een eindige verzameling van *terminalen* met $N \cap T = \emptyset$, weergegeven met kleine letters (mogelijk met subscripts en gekozen uit het begin van het alfabet);
(c) P is een eindige verzameling van *produktieregels* van de vorm $\alpha \rightarrow \beta$, waarbij α, de string uit het linkerlid van de regel zodanig is dat $\alpha \in (N \cup T)^+$ terwijl voor β geldt $\beta \in (N \cup T)^*$;
(d) $S \in N$ is een symbool dat is aangewezen als *startsymbool* voor de grammatica.

We geven een voorbeeld van een PSG. $G_1 = (\{S, A, B\}, \{a, b\}, P, S)$. P bestaat uit de produktieregels

$$\begin{array}{ll} S \rightarrow A & S \rightarrow B \\ A \rightarrow aA & A \rightarrow a \\ B \rightarrow bB & B \rightarrow b \end{array}$$

Ook hier gebruiken we het teken | voor 'of' om de produktieregels korter op te kunnen schrijven. Bovenstaand voorbeeld wordt dan

$$S \rightarrow A|B$$
$$A \rightarrow aA|a$$
$$B \rightarrow bB|b$$

Zoals u misschien al hebt opgemerkt hoeven we alleen maar de produktieregels en het startsymbool aan te geven om een grammatica te specificeren als we ons tenminste houden aan de afspraken ten aanzien van het gebruik van hoofdletters en kleine letters. Als we altijd S als startsymbool gebruiken dan hoeven we zelfs dat niet meer aan te geven.

Bekijk nu eens de volgende PSG, G_2 met produktieregels

$$
\begin{aligned}
S &\rightarrow aSBC|aBC \\
CB &\rightarrow BC \\
aB &\rightarrow ab \\
bB &\rightarrow bb \\
bC &\rightarrow bc \\
cC &\rightarrow cc
\end{aligned}
$$

Dit is een eerste voorbeeld van een grammatica met linkerleden die niet alleen uit een enkele nonterminaal bestaan. Men kan bewijzen dat uit S elke string van de vorm $a^n b^n c^n$, $n \geq 1$ kan worden afgeleid. Een voorbeeld:

$$
\begin{aligned}
S &\Rightarrow aSBC & \\
&\Rightarrow aaBCBC & \text{(als } S \rightarrow aBC) \\
&\Rightarrow aabCBC & \text{(als } aB \rightarrow ab) \\
&\Rightarrow aabbCC & \text{(als } CB \rightarrow BC) \\
&\Rightarrow aabbcC & \text{(als } bC \rightarrow bc) \\
&\Rightarrow aabbcc & \text{(als } cC \rightarrow cc)
\end{aligned}
$$

Dit is een volgens de grammatica toegelaten afleiding van $a^2b^2c^2$.

Het begrip 'afleiding' moet u nu langzamerhand intuïtief duidelijk zijn. We gaan nu onze formele definitie van het begrip grammatica gebruiken om precies te kunnen aangeven wat het betekent dat een string op grond van een grammatica afleidbaar is. Zij $G = (N, T, P, S)$ een willekeurige PSG en zij $\gamma_1\alpha\gamma_2 \in (N \cup T)^+$ een string van terminalen en nonterminalen met een lengte ≥ 1. Als $\alpha \rightarrow \beta$ een produktieregel in P is dan kan α in $\gamma_1\alpha\gamma_2$ vervangen worden door β en dit levert $\gamma_1\beta\gamma_2$ op. We schrijven dan

$$\gamma_1\alpha\gamma_2 \underset{G}{\Rightarrow} \gamma_1\beta\gamma_2,$$

(lees: $\gamma_1\alpha\gamma_2$ genereert $\gamma_1\beta\gamma_2$ of $\gamma_1\beta\gamma_2$ wordt afgeleid uit $\gamma_1\alpha\gamma_2$).

Als $\alpha_1, \alpha_2, \ldots, \alpha_n \in (N \cup T)^*$ en $\alpha_1 \underset{G}{\Rightarrow} \alpha_2, \alpha_2 \underset{G}{\Rightarrow} \alpha_3, \ldots, \alpha_{n-1} \underset{G}{\Rightarrow} \alpha_n (n > 1)$, dan schrijven we

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2 \underset{G}{\Rightarrow} \alpha_3 \cdots \underset{G}{\Rightarrow} \alpha_{n-1} \underset{G}{\Rightarrow} \alpha_n$$

of korter

$$\alpha_1 \overset{+}{\underset{G}{\Rightarrow}} \alpha_n$$

(lees: α_1 genereert α_n in één of meer stappen), $\overset{+}{\underset{G}{\Rightarrow}}$ is dus de transitieve afsluiting van de relatie $\underset{G}{\Rightarrow}$. De reflexieve afsluiting wordt aangegeven door $\overset{*}{\underset{G}{\Rightarrow}}$ en dus geldt

$$\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_n \text{ desd } \alpha_1 = \alpha_n \text{ of } \alpha_1 \overset{+}{\underset{G}{\Rightarrow}} \alpha_n.$$

Als $\alpha \in (N \cup T)^*$ zodanig is dat $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ dan heet α een *zinsvorm* van G. Een *zin van* G is iedere willekeurige zinsvorm in T^*, dat wil zeggen een terminale string die uit S kan worden afgeleid. $L(G)$ de *verzameling van alle zinnen van* G, heet de taal gegenereerd door G. Dus

$$L(G) = \{x \in T^* | S \overset{*}{\underset{G}{\Rightarrow}} x\}.$$

In veel voorbeelden is uit de context duidelijk welke grammatica G we bedoelen. In dat geval laten we het subscript G vervallen in $\underset{G}{\Rightarrow}$, $\overset{+}{\underset{G}{\Rightarrow}}$, $\overset{*}{\underset{G}{\Rightarrow}}$ en schrijven $\Rightarrow$, $\overset{+}{\Rightarrow}$, $\overset{*}{\Rightarrow}$.

Ga na dat voor de eerder gebruikte voorbeelden G_1 en G_2 geldt dat

$$L(G_1) = \{a^n | n \geq 1\} \cup \{b^n | n \geq 1\}$$

en

$$L(G_2) = \{a^n b^n c^n | n \geq 1\}$$

Soms komt het voor dat twee verschillende grammatica's G en G' dezelfde taal $L(G) = L(G')$ genereren. In dat geval heten de grammatica's *equivalent*. De grammatica G_3 met de hierna volgende produktieregels is equivalent met G_1

$$S \rightarrow aA|bB|a|b$$
$$A \rightarrow aA|a$$
$$B \rightarrow bB|b$$

2.2 Contextvrije grammatica's

De syntaxis van de meeste programmeertalen kan formeel met behulp van de BNF-notatie worden gedefinieerd. Dit komt overeen met een PSG met de beperking dat het linkerlid van elke produktieregel $\alpha \rightarrow \beta$ uit een enkele non-terminaal moet bestaan. In de formele taaltheorie heet een PSG met deze restrictie een *contextvrije* grammatica. Een contextvrije grammatica (CFG) is een PSG, $G = (N, T, P, S)$ waarbinnen elke produktie in P de volgende vorm heeft

$$A \rightarrow \beta \quad \text{waarbij} \quad A \in N \quad \text{en} \quad \beta \in (N \cup T)^*.$$

Iedere door een CFG gegenereerde taal heet een *contextvrije taal* (CFL). De term contextvrij wordt gebruikt omdat elke A in een zinsvorm kan worden uitgebreid door een produktie van de vorm $\alpha \rightarrow \beta$ te gebruiken. Deze uitbreiding is toegelaten onafhankelijk van de strings die A in de zinsvorm omringen, dus onafhankelijk van de context van A.

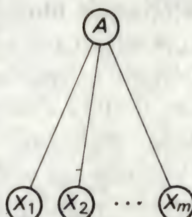
G_1 en G_3 zijn beide voorbeelden van CFG's en dus is $L = L(G_1) = L(G_3)$ een CFL. We kunnen op dit moment nog niet vaststellen of ook $L_2 = L(G_2)$

een CFL is. In ieder geval is G_2 geen contextvrije grammatica, maar misschien bestaat er toch een CFG die L_2 genereert. De lezer zou zelfs kunnen proberen om zo'n CFG te construeren, (later zullen we echter zien dat dit tijdverspilling is!). Wel kunnen we $\{a^n b^n | n \geq 1\}$ met een CFG genereren, bijvoorbeeld door de grammatica met de produktie

$$S \rightarrow aSb|ab$$

te gebruiken.

Als $G = (N, T, P, S)$ een CFG is dan kan iedere $x \in L(G)$ worden afgeleid uit het startsymbool S, zoals we zagen. De afleiding van een niet-lege string x kan handig worden weergegeven met behulp van een zogenaamde *afleidingsboom* (ook wel een *ontleedboom* en in het Engels een parse tree genaamd). De wortel van de boom krijgt het label S en de blaadjes worden van links naar rechts $a_1, a_2, \ldots, a_n$ met $a_i \in T, 1 \leq i \leq n$ en $x = a_1 a_2 \ldots a_n$. De inwendige knopen van de boom labelen we met de nonterminale symbolen die we tijdens de afleiding van x gebruiken. Als A zo'n symbool is en als dit met behulp van de produktieregel $A \rightarrow X_1 X_2 \ldots X_m (X_i \in N \cup T, i = 1, \ldots, m)$, wordt uitgebreid dan wordt knooppunt A de ouder van de m knooppunten $X_1, X_2, \ldots, X_m$, en wel zoals in figuur 2.3 is aangegeven.



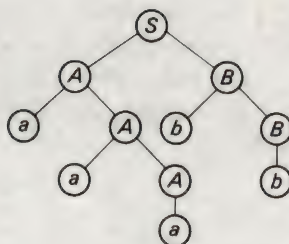
Figuur 2.3

In figuur 2.4 geven we ter illustratie de afleidingsboom die hoort bij de afleiding $a^3 b^2$ in de CFG, G_4 met de produktieregels

$$S \rightarrow AB$$
$$A \rightarrow aA|a$$
$$B \rightarrow bB|b$$

Deze afleidingsboom komt precies overeen met de volgende afleiding

$$\begin{aligned} S &\Rightarrow AB \\ &\Rightarrow aAB \\ &\Rightarrow aaAB \\ &\Rightarrow aaaB \\ &\Rightarrow aaabB \\ &\Rightarrow aaabb. \end{aligned}$$



Figuur 2.4

Voor $x \in L(G)$ zijn meestal veel verschillende afleidingen mogelijk. In G_4 bijvoorbeeld zijn dit een aantal alternatieve afleidingen voor a^3b^2:

$$S \Rightarrow AB \Rightarrow AbB \Rightarrow aAbB \Rightarrow aaAbB \Rightarrow aaAbb \Rightarrow aaabb,$$
$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aAbB \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb,$$
$$\text{en } S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaAbB \Rightarrow aaabB \Rightarrow aaabb.$$

De afleidingsboom heeft voor al deze afleidingen precies dezelfde vorm.

Als elke afleiding voor $x \in L(G)$ leidt tot dezelfde afleidingsboom dan heet de CFG, G, *ondubbelzinnig*. Bestaan er twee of meer verschillende afleidingsbomen voor een $x \in L(G)$ dan heet G *dubbelzinnig* of *ambigu*. G_4 is een voorbeeld van een ondubbelzinnige grammatica.

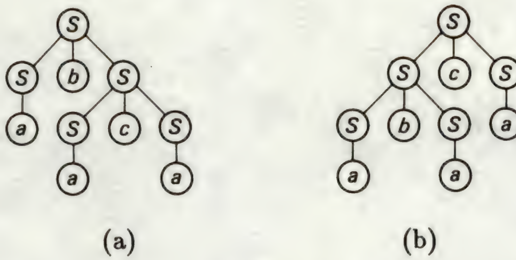
Een afleiding van $x \in L(G)$ heet een *linkerafleiding* als x steeds de meest linker nonterminaal is in de zinsvorm die wordt uitgebreid. Bij elke afleidingsboom behoort een unieke linkerafleiding en we kunnen daarom een dubbelzinnige grammatica definiëren als een CFG met een $x \in L(G)$ met twee verschillende linkerafleidingen. Zo'n dubbelzinnige taal is bijvoorbeeld G_5 met produkties

$$S \rightarrow SbS|ScS|a$$

In $L(G_5)$ zit de string *abaca* en deze heeft twee verschillende linkerafleidingen.

$$S \Rightarrow SbS \Rightarrow abS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca,$$
$$S \Rightarrow ScS \Rightarrow SbScS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca.$$

In de figuren 2.5a en 2.5b zijn de bijbehorende afleidingsbomen getekend.



Figuur 2.5

2.3 Ontleden van rekenkundige uitdrukkingen

We formuleren een eenvoudige BNF-grammatica voor het genereren van rekenkundige uitdrukkingen in de variabelen a, b en c.

<expressie> ::= <term> | <expressie> + <term> | <expressie><term>
<term> ::= <factor> | <term> × <factor> | <term> / <factor>
<factor> ::= a | b | c | (<expressie>)

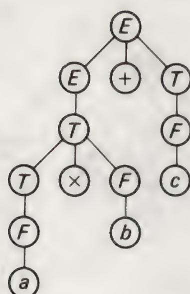
Als we E als startsymbool gebruiken dan kan deze grammatica als volgt in de formele notatie voor CFG's worden uitgeschreven

$$E \rightarrow T|E+T|E-T$$
$$T \rightarrow F|T \times F|T/F$$
$$F \rightarrow a|b|c|(E)$$

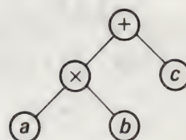
Deze grammatica is ondubbelzinnig omdat er een unieke afleidingsboom bestaat voor elke string die kan worden gegenereerd. In figuur 2.6a wordt bijvoorbeeld de enig mogelijke afleidingsboom gegeven voor de expressie $a \times b + c$. Uit de afleidingsboom kan door een compiler een zogenaamde *semantische boom* worden geconstrueerd. We geven een dergelijke boom voor de uitdrukking $a \times b + c$ in figuur 2.6b en het valt gemakkelijk in te zien hoe de boom kan worden geconstrueerd uit de bijbehorende afleidingsboom. Uit de semantische boom construeert de compiler vervolgens de machinecode voor de rekenkundige uitdrukking. Bij dit voorbeeld kan deze code er als volgt uitzien

LOAD a
MULT b
ADD c

De grammatica uit ons voorbeeld is een vereenvoudigde versie van de syntactische definitie van rekenkundige uitdrukkingen in de programmeertaal Pascal. De grammatica is niet alleen ondubbelzinnig, maar de produktieregels zijn ook



(a) Afleidingsboom



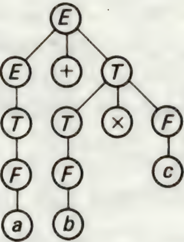
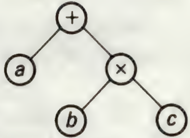
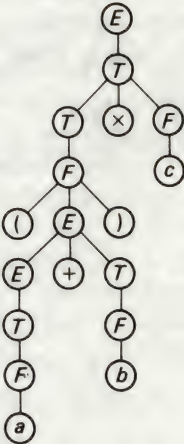
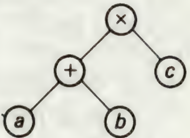
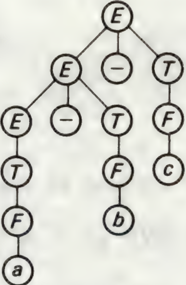
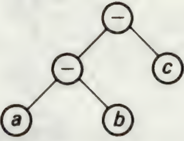
(b) Semantische boom

Figuur 2.6

zo handig geformuleerd dat elke semantische boom die uit een afleidingsboom wordt geconstrueerd ook leidt tot de gebruikelijke semantische interpretatie van de expressie. Zo liggen met name de prioriteitsregels voor rekenkundige bewerkingen (× en / eerst en vervolgens + en −) besloten in de grammatica. In figuur 2.7 hebben we nog wat voorbeelden weergegeven.
Het zou ideaal zijn als de gehele syntaxis van iedere programmeertaal zou kunnen worden gedefinieerd met behulp van CFG's. Er is veel onderzoek naar dit soort grammatica's verricht en de technieken voor het construeren van afleidingsbomen zijn volkomen helder. Als de CFG nog een paar extra eigenschappen heeft die we in volgende hoofdstukken zullen behandelen, dan kunnen efficiënte ontleedalgoritmen worden geformuleerd. Jammer genoeg kan weliswaar het grootste deel van de syntaxis van een programmeertaal op deze manier worden gedefinieerd, maar er blijven altijd een aantal extra regels over die niet zo kunnen worden uitgedrukt. Dit betekent dat de kern van een compiler wel bestaat uit een uitwerking van een ontleedalgoritme, dat leidt tot de constructie van een semantische boom, maar dat er behalve dat ook altijd bepaalde controles moeten worden uitgevoerd en dat bepaalde tabellen moeten worden bijgewerkt.

2.4 De lege string in contextvrije grammatica's

In onze definitie van CFG's beperkten we de vorm van de produktieregels tot $A \rightarrow \beta$ waarbij A een nonterminaal voorstelt en β elke willekeurige string van nonterminalen en terminalen mag zijn. Dit betekent dat we ook regels van de vorm $A \rightarrow \varepsilon$ toelaten. Dergelijke regels zullen we *ε-produkties* noemen. Dit soort produkties kan problemen geven bij het ontleden en ook bij het construeren van formele bewijzen over eigenschappen van grammatica's. Een oplettende lezer heeft zich misschien al afgevraagd hoe dergelijke produktieregels in afleidingsbomen worden weergegeven. (We laten eenvoudig ε als blaadje toe.) Een

Rekenkundige uitdrukking	Afleidingsboom	Semantische boom
$a + b \times c$		
$(a + b) \times c$		
$a - b - c$		

Figuur 2.7

grammatica zonder ε-produkties heet *ε-vrij*. Het prettigst zou zijn als al onze contextvrije grammatica's ε-vrij zouden zijn, maar als $\varepsilon \in L(G)$ voor een of andere CFG, G, dan is het onmogelijk alle ε-produkties uit G te verwijderen zonder dat daardoor de taal die door de grammatica wordt gegenereerd verandert. Het beste dat we misschien kunnen bereiken is een constructie van een CFG, G', uit G, die wel ε-vrij is, terwijl $L(G') = L(G)\backslash\{\varepsilon\}$. Gelukkig is zo'n constructie inderdaad mogelijk en niet eens erg moeilijk.

Veronderstel dat $G = (N,T,P,S)$ een CFG met ε-produkties is, dan is $G' = (N,T,P',S)$ en P' wordt als volgt uit P geconstrueerd:

1. Stop alle ε-vrije produktieregels van P in P'.
2. Bepaal alle nonterminalen $A \in N$ met $A \underset{G}{\overset{*}{\Rightarrow}} \varepsilon$. Dit soort nonterminalen noemen we *ε-genererend*. Voeg nu voor iedere $p \in P$ met één of meer ε-genererende nonterminalen in het rechterlid aan P' elke ε-vrije produktie toe die uit p kan worden geconstrueerd door één of meer van deze genererende nonterminalen weg te laten.

We zullen nu bewijzen dat $L(G') = L(G)\backslash\{\varepsilon\}$, maar eerst geven we een voorbeeld om de constructiewijze te verduidelijken. Stel dat G de volgende produktieregels heeft

$$S \rightarrow [E]|E$$
$$E \rightarrow T|E+T|E-T$$
$$T \rightarrow F|T \times F|T/F$$
$$F \rightarrow a|b|c|\varepsilon$$

dan volgt hieruit $S \overset{*}{\Rightarrow} \varepsilon, E \overset{*}{\Rightarrow} \varepsilon, T \overset{*}{\Rightarrow} \varepsilon$, en $F \overset{*}{\Rightarrow} \varepsilon$. De uit G geconstrueerde ε-vrije grammatica heeft dan de volgende produktieregels

$$S \rightarrow [E]|[\,]|E$$
$$E \rightarrow T|E+T|E-T|E+|E-|+T|-T|+|-$$
$$T \rightarrow F|T \times F|T/F|T \times |T/| \times F|/F| \times |/$$
$$F \rightarrow a|b|c$$

Nu terug naar het bewijs. We bewijzen eerst een hulpstelling.

Lemma Voor alle $A \in N$ en $x \in T^+$ geldt $A \underset{G}{\overset{*}{\Rightarrow}} x$ desd als $A \underset{G'}{\overset{*}{\Rightarrow}} x$.

Bewijs Het bewijs bestaat uit twee gedeelten.

Eerst bewijzen we dat uit $A \underset{G'}{\overset{*}{\Rightarrow}} x$ volgt dat $A \underset{G}{\overset{*}{\Rightarrow}} x$. Dit is vrij eenvoudig in te zien. Immers als $X \rightarrow \beta$ een produktie in G' is dan volgt hieruit onmiddelijk dat $X \underset{G}{\overset{*}{\Rightarrow}} \beta$. Elke afleiding binnen G' kan dus ook worden gerealiseerd in G.

Ten tweede moeten we bewijzen dan uit $A \underset{G}{\overset{*}{\Rightarrow}} x$ volgt dat $A \underset{G'}{\overset{*}{\Rightarrow}} x$ voor alle $A \in N$ en $x \in T^+$. Dit doen we door middel van inductie naar n, het

aantal stappen in de afleiding van x uit A in de grammatica G. Als $n = 0$ is het te bewijzen triviaal. Als $n = 1$ bestaat er een directe produktieregel $A \rightarrow x$ in G en omdat $x \neq \varepsilon$ bestaat er dan ook zo'n regel in G'. Hieruit volgt $A \underset{G'}{\Rightarrow} x$. Veronderstel nu dat het te bewijzen geldt voor alle afleidingen van een niet-lege terminale string uit een nonterminaal in k of minder stappen en stel dat $A \overset{*}{\underset{G}{\Rightarrow}} x$, $k+1$ stappen vergt. Er bestaat dan een produktie in P van de vorm $A \rightarrow X_1, X_2, \ldots, X_m$, $X_i \in N \cup T$, met $A \underset{G}{\Rightarrow} X_1X_2 \ldots X_m \overset{*}{\underset{G}{\Rightarrow}} x$. De terminale string x kan worden verdeeld in substrings $x = x_1x_2 \ldots x_m$ zodanig dat uit $X_i \in N$ volgt dat $X_i \overset{*}{\underset{G}{\Rightarrow}} x_i$, en als $X_i \in T$ dan volgt $X_i = x_i$ (en dus per definitie $X_i \overset{*}{\underset{G}{\Rightarrow}} x_i$). Enkele van de strings x_i $(1 \leq i \leq m)$ kunnen leeg zijn omdat het ε-produkties in G zijn. Veronderstel dat de indices $1 \leq i1 < i2 < \ldots < ir \leq m$ de niet-lege strings $x_{i1}, x_{i2}, \ldots x_{ir}$ aangeven. Dan geldt $x = x_{i1}x_{i2} \ldots x_{ir}$ en $X_{ij} \overset{*}{\underset{G}{\Rightarrow}} x_{ij}$, $j = 1, \ldots, r$. Al deze afleidingen kosten k of minder stappen en dus weten we op grond van onze inductieveronderstelling dat $X_{ij} \overset{*}{\underset{G'}{\Rightarrow}} x_{ij}$, $j = 1, \ldots, r$. Omdat $A \rightarrow X_{i1}X_{i2} \ldots X_{ir}$ een produktie in G' is, volgt dat $A \overset{*}{\underset{G'}{\Rightarrow}} x$ en dit geldt voor alle afleidingen van $k+1$ stappen. Uit inductie naar k volgt nu dat het te bewijzen geldt voor alle afleidingen.

Uit dit lemma volgt onmiddellijk dat $x \in T^+$, $S \overset{*}{\underset{G}{\Rightarrow}} x$, desd $S \overset{*}{\underset{G'}{\Rightarrow}} x$. Dus geldt $x \in L(G) \backslash \{\varepsilon\}$ desd als $x \in L(G')$ en hiermee hebben we de volgende stelling bewezen.

Stelling 2.1

Voor elke CFL, L, bestaat er een ε-vrije CFG, G, zodat $L(G) = L \backslash \{\varepsilon\}$.

Als in het vervolg voor één of andere CFL, L geldt dat $\varepsilon \in L$ dan zullen we eisen dat de enige produktie in een CFG, G, die L genereert van de vorm $S \rightarrow \varepsilon$ is. We mogen in dat geval ook aannemen dat S niet in het rechterlid van enige andere produktie voorkomt. Dit geldt op grond van de volgende redenering. Zij $G = (N, T, P, S)$ een ε-vrije grammatica die $L \backslash \{\varepsilon\}$ genereert. Zij nu S' een nieuw nonterminaal symbool en definieer $G' = (N \cup \{S'\}, T, P \cup \{S' \rightarrow \varepsilon | S\}, S')$. G' heeft nu alle gewenste eigenschappen en $L(G') = L$.

2.5 Oefeningen

1. Beschrijf contextvrije grammatica's voor de volgende talen:

   (a) alle strings in $\{0,1\}^*$, waarbij na elke 0 onmiddellijk een 1 volgt;

   (b) alle strings in $\{0,1\}^*$ die palindromen zijn. Een palindroom is een string die gelijk blijft als men hem omkeert;

   (c) alle strings in $\{0,1\}^*$ met twee keer zoveel nullen als enen.

2. Bekijk de CFG met produktieregels

$$S \rightarrow AB$$
$$A \rightarrow SA|BB|bB$$
$$B \rightarrow b|aA|\varepsilon$$

Ontwerp een equivalente grammatica met alleen de ε-produktie $S \rightarrow \varepsilon$.

3. Bewijs dat de grammatica met als produktieregels

$$S \rightarrow bA|aB$$
$$A \rightarrow a|aS|bAA$$
$$B \rightarrow b|bS|aBB$$

de taal in $\{a, b\}^*$ genereert die bestaat uit strings met evenveel a's als b's. (Hint: bewijs met inductie dat voor elke zinsvorm de som van het aantal a's en A's gelijk is aan die van het aantal b's en B's.)

4. Bewijs dat $L(G_2) = \{a^n b^n c^n | n \geq 1\}$. G_2 is de voorbeeld-PSG uit dit hoofdstuk.

5. Bewijs dat de grammatica uit oefening 2.3 dubbelzinnig is. Bewijs door uit te gaan van de grammatica met de volgende produktieregels

$$S \rightarrow aBS|aB|bAS|bA$$
$$A \rightarrow bAA|a$$
$$B \rightarrow aBB|b$$

dat er ook een ondubbelzinnige grammatica voor dezelfde taal bestaat. (Dit is niet altijd het geval. Er bestaan *inherent dubbelzinnige context-vrije talen*. Voor deze talen bestaan geen ondubbelzinnige grammatica's.)

6. Als elke produktie in een grammatica G die geen ε-produktie is, de vorm $A \rightarrow aB$ of $A \rightarrow a$ heeft,waarbij $A, B \in N$ en $a \in T$, dan heet G een *reguliere grammatica*. Bewijs dat de ε-vrije grammatica G' die in het bewijs van stelling 2.1 uit G werd geconstrueerd regulier is als G regulier is.

7. Toon met behulp van de Pascal-syntaxis aan dat de volgende teksten toegelaten Pascal-programma's zijn.

(a)
```
program ex(output);
begin
    if 1 + 1 = 2 then write ('hoera')
end.
```

(b)
```
program ex2;
begin
    write(1 + 1)
end.
```

(c)
```
program copy(f1, f2);
    var f1, f2: file of char;
    begin reset (f1); rewrite (f2);
            while not eof (f1) do
                    begin f2↑ := f1; put (f2); get (f1);
                    end
    end.
```

8. Construeer een Pascal-programma dat syntactisch correct is maar dat niet door een Pascal-compiler wordt geaccepteerd.

Hoofdstuk 3

Reguliere talen

Moeten kiezen maakt het leven moeilijk.
—GEORGE MOORE
The Bending of the Bough

Tekenloze gehele getallen kunnen als volgt in BNF worden gedefinieerd.

```
<getallenrij> ::= 0|1|2|3|4|5|6|7|8|9
        |0 <getallenrij> |1 <getallenrij> |2 <getallenrij>
        |3 <getallenrij> |4 <getallenrij> |5 <getallenrij>
        |6 <getallenrij> |7 <getallenrij> |8 <getallenrij>
        |9 <getallenrij>
<tekenloos geheel getal> ::= 0|1|2|3|4|5|6|7|8|9
        |1 <getallenrij> |2 <getallenrij> |3 <getallenrij>
        |4 <getallenrij> |5 <getallenrij> |6 <getallenrij>
        |7 <getallenrij> |8 <getallenrij> |9 <getallenrij>
```

Elk rechterlid in dit voorbeeld is óf een eenvoudig terminaal symbool, óf een terminaal gevolgd door een nonterminaal. Alle elementaire symbolen in programmeertalen (gehele getallen, namen van variabelen, operatoren, gereserveerde woorden, leestekens, enzovoort) kunnen met dit type regels worden gedefinieerd. Veel van de tijd die nodig is om een programma te compileren wordt besteed aan het herkennen van dit soort elementaire symbolen en er is daarom zeker reden om grammatica's met produktieregels van deze vorm nader te bestuderen. Dit type grammatica's heet regulier en de talen die erdoor worden gedefinieerd heten reguliere talen. In dit hoofdstuk zullen we zien dat reguliere talen op een heel efficiënte manier kunnen worden herkend en dat hun grammatica's veel gunstige eigenschappen bezitten. Jammer genoeg zijn reguliere grammatica's erg beperkt in hun mogelijkheden; je kunt er zelfs de eenvoudigste programmaconstructies niet in beschrijven. Tijdens de compilatie kunnen ze alleen gebruikt worden voor het herkennen van de elementaire symbolen in een programma; dit heet de *scanning fase* of de *lexicale analyse*. Als deze fase klaar is dan zijn er verfijndere technieken nodig om het programma verder te ontleden. Enkele van deze technieken zullen we in latere hoofdstukken tegenkomen.

3.1 Reguliere grammatica's

Een frasestructuurgrammatica $G = (N, T, P, S)$ heet een *reguliere grammatica* onder de volgende voorwaarden:
(i) als er een ε-produktie in G voorkomt dan is die van de vorm $S \rightarrow \varepsilon$ en dan verschijnt S niet als een substring in het rechterlid van enig andere produktie in P;
(ii) alle andere produktieregels hebben de vorm

$$A \rightarrow a \text{ met } A \in N, a \in T$$

of

$$A \rightarrow aB \text{ met } A, B \in N \text{ en } a \in T$$

Een taal heet een *reguliere taal* dan en slechts dan als zij door een reguliere grammatica wordt voortgebracht. Uit de definitie en de opmerkingen na stelling 2.1 is het duidelijk dat L een reguliere taal is desd als $L \backslash \{\varepsilon\}$ wordt gegenereerd door een ε-vrije reguliere grammatica.

De reguliere grammatica G_1 met de produktieregels

$$S \rightarrow aS|aB$$
$$B \rightarrow bB|b$$

genereert de reguliere taal $L(G_1) = \{a^m b^n | m, n \geq 1\}$. Dit specifieke voorbeeld is ε-vrij omdat $\varepsilon \notin L(G_1)$. De reguliere taal $L = L(G_1) \cup \{\varepsilon\}$ wordt gegenereerd door de reguliere grammatica met produktieregels

$$S \rightarrow \varepsilon|aS_1|aB$$
$$S_1 \rightarrow aS_1|aB$$
$$B \rightarrow bB|b$$

Elke produktieregel in een reguliere grammatica vervangt een nonterminaal door een string die hoogstens één nonterminaal bevat. Hieruit volgt dat elke zinsvorm ofwel in T^* zit ofwel precies één element van N bevat. Verder is noodzakelijkerwijs elke nonterminaal in een zinsvorm het meest rechter symbool in die zinsvorm.

Hebben we eenmaal een ε-vrije reguliere grammatica $G = (N, T, P, S)$ die L genereert dan kunnen we gemakkelijk een ε-vrije reguliere grammatica $G^+ = (N, T, P^+, S)$ construeren die L^+ genereert. We doen dat door produkties P^+ uit P te construeren door steeds produkties van de vorm $A \rightarrow a$ in P te vervangen door produkties van de vorm $A \rightarrow aS$. G_1 leidt dan tot G_1^+ met produktieregels

$$S \rightarrow aS|aB$$
$$B \rightarrow bB|b|bS$$

Als we terugkeren tot het algemene geval dan moeten we bewijzen dat $L(G^+) = L^+$, dus dat $L(G^+) \subset L^+$ en dat $L^+ \subset L(G^+)$. We bewijzen eerst dat $L(G^+) \subset L^+$. Als $x \in L(G^+)$ dan zullen we door middel van inductie naar n (het aantal 'nieuwe' produkties in $P^+ \backslash P$ dat we bij de afleiding van x gebruiken) bewijzen dat $x \in L^+$. Als $n = 0$ dan werden bij de afleiding alleen produkties uit P gebruikt en dus is $x \in L$ en dus ook $x \in L^+$. Veronderstel nu dat als $n < k$ geldt dat $x \in L^+$ en ga uit van een afleiding van $x \in L(G^+)$ die k nieuwe produktieregels gebruikt. Stel dat $A \rightarrow aS$ de laatste van die produktieregels is dan volgt $S \overset{+}{\underset{G^+}{\Rightarrow}} yA \underset{G^+}{\Rightarrow} yaS \overset{+}{\underset{G^+}{\Rightarrow}} yaz = x$, met $y, z \in T^+$ en $a \in T$. $yaS \overset{+}{\underset{G^+}{\Rightarrow}} yaz$ vergt geen nieuwe produkties en dus volgt $yaS \overset{+}{\underset{G}{\Rightarrow}} yaz$ en dus $S \overset{+}{\underset{G}{\Rightarrow}} z$, dat wil zeggen dat $z \in L$. Ook geldt, omdat $A \rightarrow aS$ een nieuwe produktieregel is, dat er een produktie $A \rightarrow a$ in P moet bestaan. Dus is $S \overset{+}{\underset{G^+}{\Rightarrow}} yA \underset{G}{\Rightarrow} ya$ een afleiding van $ya \in L(G^+)$ waarvoor minder dan k nieuwe produktieregels nodig zijn. Uit de inductieveronderstelling volgt nu dat $ya \in L^+$. Als $ya \in L^+$ en $z \in L$ dan volgt hieruit dan $x = yaz \in L^+$ en hiermee hebben we het eerste deel bewezen.

Vervolgens bewijzen we dan $L^+ \subset L(G^+)$. Als $x \in L^+$ dan is $x \in L^n$ voor een $n \geq 1$. We bewijzen met inductie dat uit $x \in L^n$ volgt dat $x \in L(G^+)$. Als $n = 1$ dan geldt $x \in L$ en dus kan x uit S worden gegenereerd door alleen gebruik te maken van produkties in G. Al deze produkties bevinden zich ook in P^+ en dus is de afleiding ook binnen G^+ toegelaten. Hieruit volgt dat $x \in L(G^+)$. Veronderstel dat het te bewijzen geldt voor alle $n < k$ en ga uit van $x \in L^k$. Dit wil zeggen dat $x = yz$ met $y \in L$ en $z \in L^{k-1}$. Zij $S \overset{+}{\underset{G}{\Rightarrow}} y$ een afleiding van y in G. De laatste produktieregel die bij deze afleiding werd gebruikt is van de vorm $A \rightarrow a$ met $y = y'a$ en $y' \in T^*$. $A \rightarrow aS$ is dus een produktie in P^+ en dus is yS een zinsvorm uit G^+ en hieruit volgt weer dat $yz \in L(G^+)$. Hiermee is het tweede deel bewezen.

Als L een willekeurige reguliere taal is dan is $L^* = (L\backslash\{\varepsilon\})^+ \cup \{\varepsilon\}$. We hebben aangetoond dat $(L\backslash\{\varepsilon\})^+$ een reguliere taal is en dus geldt dit ook voor L^*. Hiermee hebben we het eerste gedeelte van de volgende stelling bewezen.

Stelling 3.1

(a) Als L een reguliere taal is dan is de Kleene afsluiting van L, L^* dit ook.
(b) Als L_1 en L_2 reguliere talen zijn dan geldt dit ook voor $L_1 \cup L_2$ en L_1L_2.

Het bewijs van het tweede gedeelte van de stelling berust ook op het construeren van nieuwe grammatica's uit oude. We geven hier een schets van die constructies, maar laten de invulling van het bewijs aan de lezer over.

Bij het bewijs van de eigenschap voor de vereniging mogen we veronderstellen dat L_1 en L_2 beide ε niet bevatten. (Als $\varepsilon \in L_1$ of $\varepsilon \in L_2$ dan $\varepsilon \in L_1 \cup L_2$ en dan kan de grammatica die $L_1 \cup L_2$ genereert gemakkelijk worden geconstrueerd uit een grammatica die $(L_1\backslash\{\varepsilon\}) \cup (L_2\backslash\{\varepsilon\})$ genereert.) Voordat we de constructie beschrijven geven we eerst een voorbeeld. Zij $L_1 = \{a^m b^n | m, n \geq 1\}$ de taal die wordt gegenereerd door de grammatica G_1 die we hiervoor beschreven

en stel dat L_2 wordt gegenereerd door een grammatica G_2 met produktieregels

$$S \rightarrow cS|c$$

Dus is $L_2 = \{c^n | n \geq 1\}$. Onze eerste poging om een grammatica te definiëren die $L_1 \cup L_2$ genereert, zou er uit kunnen bestaan eenvoudig alle produktieregels van G_1 en van G_2 uit te schrijven. Dit is echter een foute benadering omdat de door elkaar lopende produktieregels strings $\notin L_1 \cup L_2$ zouden kunnen genereren. Zo zouden we in ons voorbeeld kunnen krijgen

$$S \Rightarrow aS \Rightarrow ac$$

We moeten er daarom voor zorgen dat elke afleiding ofwel alleen produktieregels uit G_1, ofwel alleen uit G_2 gebruikt. Om dat mogelijk te maken moeten we de nonterminale symbolen uit G_1 en G_2 van elkaar kunnen onderscheiden. We herschrijven daartoe $G_1 = (\{S_1, B_1\}, \{a, b\}, P_1, S_1)$ met een verzameling produktieregels P_1 gedefinieerd door

$$\begin{aligned} S_1 &\rightarrow aS_1|aB_1 \\ B_1 &\rightarrow bB_1|b \end{aligned}$$

en $G_2 = (\{S_2\}, \{c\}, P_2, S_2)$ met P_2 gedefinieerd door

$$S_2 \rightarrow cS_2|c$$

S_1, B_1 en S_2 zijn verschillende symbolen. Nu willen we alle produktieregels combineren en óf S_1 óf S_2 als startsymbool kiezen. Er kan maar één startsymbool zijn, laten we dat dus S noemen. De regel $S \rightarrow S_1|S_2$ mogen we niet opschrijven, want dit soort regels is in reguliere grammatica's niet toegelaten. Wat we wel kunnen doen is gebruik maken van produkties van de vorm $S \rightarrow \alpha$ voor alle α zodanig dat of $S_1 \rightarrow \alpha$ een produktie in G_1 is of $S_2 \rightarrow \alpha$ een produktie in G_2. De reguliere grammatica voor ons voorbeeld heeft dan de volgende produktieregels

$$\begin{aligned} S &\rightarrow aS_1|aB_1|cS_2|c \\ S_1 &\rightarrow aS_1|aB_1 \\ B_1 &\rightarrow bB_1|b \\ S_2 &\rightarrow cS_2|c \end{aligned}$$

Het is vrij eenvoudig in te zien dat we met deze grammatica strings van de vorm $\{a^m b^n | m, n \geq 1\} \cup \{c^n | n \geq 1\}$ genereren.

Na deze overwegingen kunnen we de formele constructiewijze aangeven. Stel $G_1 = (N_1, T_1, P_1, S_1)$ en $G2 = (N_2, T_2, P_2, S_2)$ zijn ε-vrije reguliere grammatica's die respectievelijk de talen L_1 en L_2 genereren. Zonder verlies van algemeenheid mogen we veronderstellen dat $N_1 \cap N_2 = \emptyset$. Zij $S \notin N_1 \cup N_2$ een nieuw symbool en construeer $G = (N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup P_3, S)$, waarbij P_3 de verzameling van alle produktieregels $S \rightarrow \alpha$ voorstelt met $S_1 \rightarrow \alpha$ in

P_1 of $S_2 \to \alpha$ in P_2. Het formele bewijs dat $L(G) = L(G_1) \cup L(G_2)$ wordt aan de lezer overgelaten.

We kunnen ook een grammatica G' uit G_1 en G_2 construeren zodanig dat $L(G') = L(G_1)L(G_2)$. Als we bedenken dat $N_1 \cap N_2 = \emptyset$, dan is $G' = (N_1 \cup N_2, T_1 \cup T_2, P', S_1)$. Hierbij is P' de verzameling van alle produkties van P_1 en P_2, behalve dat elke produktie uit P_1 van de vorm $A \to a$ is vervangen door $A \to aS_2$. Opnieuw is het aan de lezer een en ander formeel te bewijzen. De constructie leidt bijvoorbeeld tot de volgende verzameling van produktieregels

$$S_1 \to aS_1|aB_1$$
$$B_1 \to bB_1|bS_2$$
$$S_2 \to cS_2|c$$

3.2 Eindige automaten

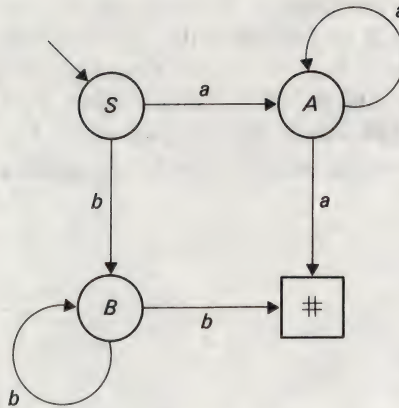
Elke reguliere grammatica $G = (N, T, P, S)$ kan worden voorgesteld door een gerichte graaf met gelabelde kanten en knopen. De knopen in de graaf zijn allen verschillend en gelabeld met verschillende elementen uit N. Verder is er een speciale 'halt'-knoop met het teken # als label. Het is gebruikelijk de knoop met het startsymbool S van een extra pijl te voorzien terwijl de haltknoop als een vierkantje wordt getekend. Als er een produktie $A \to aB$ in P voorkomt dan wordt knoop A verbonden met knoop B door een gerichte kant met als label a. Komt er een regel $A \to a$ voor in P dan wordt A met een gerichte kant a verbonden met de haltknoop #. Elke kant in de graaf komt zo overeen met één en niet meer dan één produktie in P.

De reguliere grammatica G_3 met produktieregels

$$S \to aA|bB$$
$$A \to aA|a$$
$$B \to bB|b$$

wordt dan weergegeven door de gelabelde digraaf uit figuur 3.1

Loop nu eens langs een willekeurig pad van de startknoop S naar de haltknoop (of eindknoop) #. De labels van de bij dit pad behorende kanten vormen samen een string. De verzameling van al dergelijke strings is juist de taal $L(G_3)$ en elk pad komt op natuurlijke wijze overeen met een afleiding. Zo komt het pad van Ⓢ naar Ⓐ, weer naar Ⓐ, nog eens naar Ⓐ en vervolgens naar [#] overeen met de afleiding $S \Rightarrow aA \Rightarrow aaA \Rightarrow aaaA \Rightarrow aaaa$.



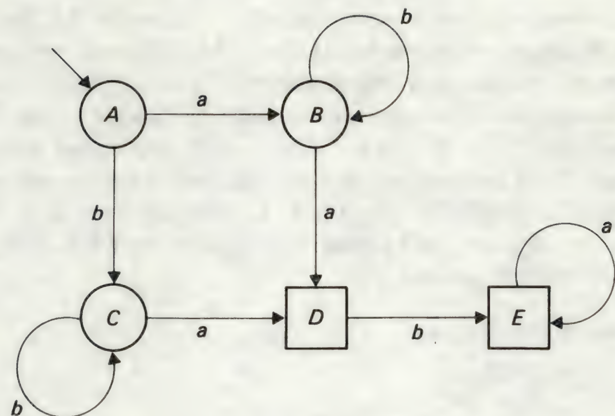
Figuur 3.1

Een gelabelde digraaf met een vaste startknoop en één (of meer) eindknopen beschrijft een eindige automaat (ook wel een eindige toestandsmachine genoemd). We beperken ons om te beginnen tot *deterministische* eindige automaten, waarbij hoogstens één kant met hetzelfde label elke knoop verlaat. Het voorbeeld in figuur 3.1 is nondeterministisch omdat er twee kanten a uit knoop A vertrekken.

Een deterministische eindige automaat (Engels: *deterministic finite state automaton* of DFSA) is een vijftupel $M = (K, T, t, k_1, F)$ met

(a) K is een eindige verzameling *toestanden*,
(b) T is een eindig *invoeralfabet*,
(c) t is een (mogelijk partiële) *overgangsfunctie* $K \times T \to K$ die aangeeft welke de volgende toestand zal zijn gegeven de huidige toestand en de input,
(d) $k_1 \in K$ is de *begintoestand*,
(e) $F \subset K$ is een verzameling *eindtoestanden*. (Noot: elke toestand in K kan worden aangewezen als eindtoestand.)

Grafisch zullen we DFSA's weergeven als gelabelde digrafen met een pijl verwijzend naar de begintoestand k_1 en met vierkantjes voor de eindtoestanden $k \in F$. In figuur 3.2 ziet u een voorbeeld. Als vijf-tupel kunnen we deze DFSA beschrijven als $(\{A, B, C, D, E\}, \{a, b\}, t, A, \{D, E\})$ met de volgende tabel ter definitie van de overgangsfunctie t:



Figuur 3.2

toestand (K)	input (T)	volgende toestand (K)
A	a	B
A	b	C
B	a	D
B	b	B
C	a	D
C	b	C
D	b	E
E	a	E

In dit geval is t een partiële functie omdat noch $t(D, a)$, noch $t(E, b)$ gedefinieerd zijn. We kunnen t ook weergeven met een zogenaamd *overgangsarray*. Als $M = (K, T, t, k_1, F)$ een* DFSA is, dan is het overgangsarray dat t definieert een matrix A_t met dimensies $\#(K) \times \#(T)$. De elementen van A_t zijn gehele getallen. Noemen we de toestanden $k_1, k_2, \ldots, k_n$ en het inputalfabet $a_1, a_2, \ldots, a_m$ dan is $A_t(i, j) = l$ desd als $t(k_1, a_j) = k_l$. Als we in ons voorbeeld de toestanden en de inputsymbolen alfabetisch ordenen dan wordt de overgangsmatrix

$$\begin{pmatrix} 2 & 3 \\ 4 & 2 \\ 4 & 3 \\ . & 5 \\ 5 & . \end{pmatrix}$$

Als t, zoals in dit voorbeeld, partieel is, dan zijn niet alle elementen uit de matrix gedefinieerd.

We zijn geïnteresseerd in de strings die bestaan uit labels langs de paden vanuit de starttoestand naar de eindtoestanden van een DFSA. De verzameling van al dergelijke strings voor een bepaalde DFSA, M, wordt weergegeven door $T(M)$ en heet de *verzameling van strings die door M worden geaccepteerd.* Om dit formeel te definiëren moeten we eerst de functie t van $K \times T \to K$ uitbreiden naar $K \times T^* \to K$. Als $k \in K$ en $x \in T^*$ dan willen we dat $t(k, x)$ de toestand voorstelt die we bereiken als we in toestand k zijn en een pad x volgen. Als $x = \varepsilon$ dan is duidelijk dat $t(k, \varepsilon) = k$. Anders is $x = ay$ voor een $a \in T$ en een $y \in T^*$ en dan kunnen we t recursief definiëren door $t(k, x) = t(t(k, a), y)$.

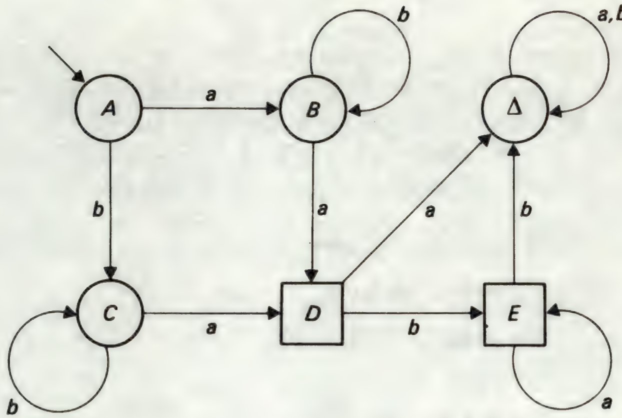
In ons voorbeeld krijgen we dan

$$\begin{aligned} t(A, aba) &= t(t(A, a), ba) \\ &= t(B, ba) \\ &= t(t(B, b), a) \\ &= t(B, a) = D. \end{aligned}$$

Nu kunnen we de verzameling strings $T(M) \subset T^*$ definiëren die wordt geaccepteerd door een DFSA, $M = (K, T, t, k_1, F)$:

$$T(M) = \{x \in T^* | t(k_1, x) \in F\}.$$

Als u bezwaar heeft tegen het feit dat de overgangsfunctie t in een DFSA, M, partieel kan zijn, dan is dit op te lossen door het toevoegen van een dummy-toestand Δ. Ongedefinieerde waarden $t(k, a)$ kan men nu de waarde $t(k, a) = \Delta$ geven en verder stellen we ook $t(\Delta, a) = \Delta$ voor alle $a \in T$. De DFSA uit ons eerdere voorbeeld gaat er dan uitzien als in figuur 3.3 is getekend. (De twee labels a, b bij de kant vanuit de dummy-toestand is een verkorte weergave van twee kanten die respectievelijk a en b gelabeld zijn.) Wellicht bezwaarlijker is de beperking dat de automaat deterministisch moet zijn. Onze volgende stap is daarom om meer dan één kant met een gegeven label vanuit een zelfde knoop toe te laten. Als nu de overgangsfunctie t wordt toegepast op $K \times T$ levert dit een verzameling van mogelijke nieuwe toestanden in plaats van maar één toestand. Als er geen volgende toestanden bestaan voor een bepaald paar $(k, a) \in K \times T$ dan is $t(k, a) = \emptyset$ en omdat $\emptyset \in 2^K$ (2^K stelt de verzameling van alle mogelijke deelverzamelingen van K voor; de machtsverzameling), volgt dat $t : K \times T \to 2^K$ altijd een totale functie is.



Figuur 3.3

Een *nondeterministische eindige automaat* (NFSA) is een vijf-tupel $M = (K, T, t, k_1, F)$ met

(a) K is een eindige verzameling *toestanden*,
(b) T is een eindig *inputalfabet*,
(c) t is een totale functie $K \times T \to 2^K$, de *overgangsfunctie* genaamd,
(d) $k_1 \in K$ is de tevoren aangewezen *begintoestand*,
(e) $F \subset K$ is een verzameling *eindtoestanden*.

De eindige automaat in figuur 3.1 was, zoals we eerder zagen, non-deterministisch. Haar overgangsfunctie wordt gedefinieerd door de volgende tabel

K	T	2^k
S	a	$\{A\}$
S	b	$\{B\}$
A	a	$\{A, \#\}$
A	b	$\emptyset$
B	a	$\emptyset$
B	b	$\{B, \#\}$
$\#$	a	$\emptyset$
$\#$	b	$\emptyset$

Voordat we de verzameling strings kunnen definiëren die door een NFSA wordt geaccepteerd moeten we weer de overgangsfunctie t uitbreiden. Allereerst laten we t opereren op $2^K \times T$ door te definiëren

$$t(K',a) = \bigcup_{k \in K'} t(k,a)$$

voor $K' \subset K, a \in T$.

Als de automaat in een van de toestanden in K' is en we volgen een kant met label $a \in T$ dan is de verzameling toestanden die we kunnen bereiken bepaald door $t(K', a)$. We breiden t nu opnieuw uit, deze keer tot een functie op $2^K \times T^* \to 2^K$. We gebruiken hierbij dezelfde methode als eerder bij de DFSA's. Als $K' \subset K$ dan is $t(K', \varepsilon) = K'$ en als $x = ay, a \in T$ en $y \in T^*$ dan is $t(K', x) = t(t(K', a), y)$.

Als we de NFSA uit figuur 3.1 gebruiken dan krijgen we bijvoorbeeld

$$\begin{aligned} t(\{S,A\},ab) &= t(t(\{S,A\},a),b) \\ &= t(t(S,a) \cup (A,a),b) \\ &= t(\{A,\#\},b) \\ &= t(A,b) \cup (\#,b) = \emptyset. \end{aligned}$$

Voor een willekeurige NFSA $M = (K, T, t, k_1, F)$ definiëren we nu $T(M)$, *de verzameling strings die door M worden geaccepteerd*, als juist die strings die overeenkomen met de labels op de paden die van een begintoestand naar een eindtoestand leiden. Formeler,

$$T(M) = \{x \in T^* | t(\{k_1\}, x) \cap F \neq \emptyset\}.$$

Het lijkt er op het eerste gezicht misschien op dat NFSA's veel krachtiger mechanismen zijn dan DFSA's, maar dit is merkwaardigerwijs niet het geval.

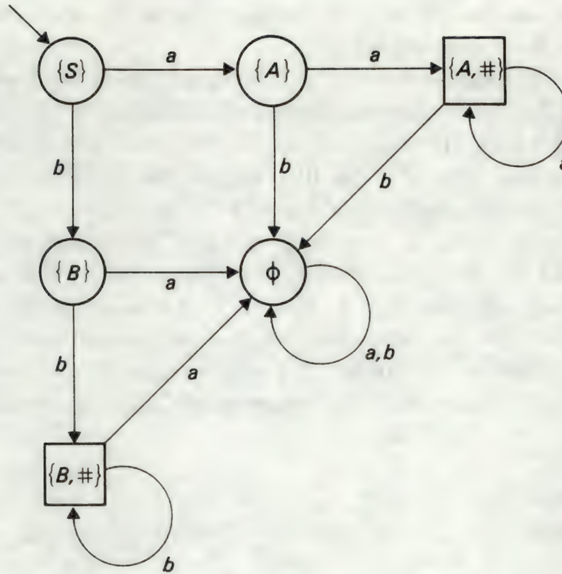
Stelling 3.2

L wordt geaccepteerd door een DFSA dan en slechts dan als L wordt geaccepteerd door een NFSA.

Bewijs:

$\Rightarrow$: Bij elke DFSA bestaat er een equivalente DFSA met een totale overgangsfunctie die gemakkelijk kan worden geconstrueerd door het toevoegen van een dummy-toestand. We veronderstellen dus dat $L \subset T^*$ zodanig is dat $L = T(M)$ voor een of andere DFSA, $M = (K, T, t, k_1, F)$, waarbij t een totale functie is. We kunnen van zo'n DFSA eenvoudig een NFSA $M' = (K, T, t', k_1, F)$ maken door $t'(k, a) = \{t(k, a)\}$ te definiëren. Uit de definities volgt dan dat $T(M) = T(M')$.

$\Leftarrow$: Stel $M = (K, T, t, k_1, F)$ is een NFSA. We gaan nu een DFSA construeren die $T(M)$ accepteert. Deze DFSA heeft als toestandsverzameling de machtsverzameling 2^K, dat wil zeggen de verzameling van alle mogelijke deelverzamelingen van K. Verder een inputalfabet T, een begintoestand $\{k_1\}$ en een overgangsfunctie gedefinieerd als de uitbreiding van t op $2^K \times T \to 2^K$. Per definitie geldt nu $x \in T(M)$ desd als $t(\{k_1\}, x) \cap F \neq \emptyset$. Als we dus de eindtoestanden van de DFSA precies gelijk maken aan die verzamelingen $K' \subset K$, waarvoor geldt $K' \cap F \neq \emptyset$, dan is hiermee de constructie voltooid.



Figuur 3.4

In de praktijk is het niet erg moeilijk om een NFSA om te zetten in een equivalente DFSA, die dus dezelfde taal accepteert. We bepalen onze aandacht om te beginnen tot de toestanden die uit de begintoestand $\{k_1\}$ kunnen worden bereikt. Dus in plaats van alle $2^{\#(K)}$ mogelijke toestanden te bekijken, beginnen we eenvoudig met $\{k_1\}$ en berekenen $t(\{k_1\}, a)$ voor alle $a \in T$. Zo krijgen we een aantal nieuwe toestanden die met $\{k_1\}$ direct verbonden zijn. Voor elke toestand K' berekenen we vervolgens $t(K', a)$ voor alle $a \in T$ en we voeren zonodig nieuwe toestanden in. Dit proces herhalen we totdat er geen nieuwe toestanden meer worden ingevoerd. Omdat er maximaal $2^{\#(K)}$ toestanden kunnen worden geconstrueerd weten we zeker dat het constructieproces eindig is. In figuur 3.4 is de DFSA getekend die equivalent is met de NFSA uit figuur 3.1. De toestand met label $\emptyset$ is in feite een dummy-toestand en kan samen met alle kanten die er heen leiden worden verwijderd.

We kunnen nu de belangrijkste stelling over eindige automaten bewijzen.

Stelling 3.3

De volgende beweringen zijn gelijkwaardig:

(i) L wordt geaccepteerd door een NFSA
(ii) L wordt geaccepteerd door een DFSA
(iii) L wordt gegenereerd door een reguliere grammatica.

Bewijs:
In stelling 3.2 toonden we al aan dat bewering (i) en (ii) equivalent zijn. Hier zullen we bewijzen dat (ii) $\Rightarrow$ (iii) en dat (iii) $\Rightarrow$ (i). Het volledige formele bewijs laten we aan de lezer over.

(ii) $\Rightarrow$ (iii): Veronderstel dat L geaccepteerd wordt door een DFSA $M = (K, T, t, k_1, F)$. Zonder verlies van algemeenheid mogen we aannemen dat $K \cap T = \emptyset$. We construeren nu een reguliere grammatica G met $L(G) = T(M) \backslash \{\varepsilon\}$. (Als $\varepsilon \in T(M)$ dan is $k_1 \in F$ en dan kunnen we gemakkelijk een reguliere grammatica G' uit G construeren met $L(G') = L(G) \cup \{\varepsilon\} = T(M)$.) De grammatica die we nodig hebben is $G = (N, T, P, S)$ met $N = K$, $S = k_1$ terwijl P bestaat uit alle produkties $k_i \rightarrow ak_j$ met $t(k_i, a) = k_j$ samen met alle produkties $k_i \rightarrow a$ met $t(k_i, a) = k_j$ en $k_j \in F$. Als we bijvoorbeeld de toestanden uit figuur 3.4 opnieuw benoemen en de dummy-toestanden weglaten, dan krijgen we de DFSA uit figuur 3.5. Bij deze automaat hoort een grammatica met toestanden $\{k_1, k_2, k_3, k_4, k_5\}$, begintoestand k_1 en produktieregels

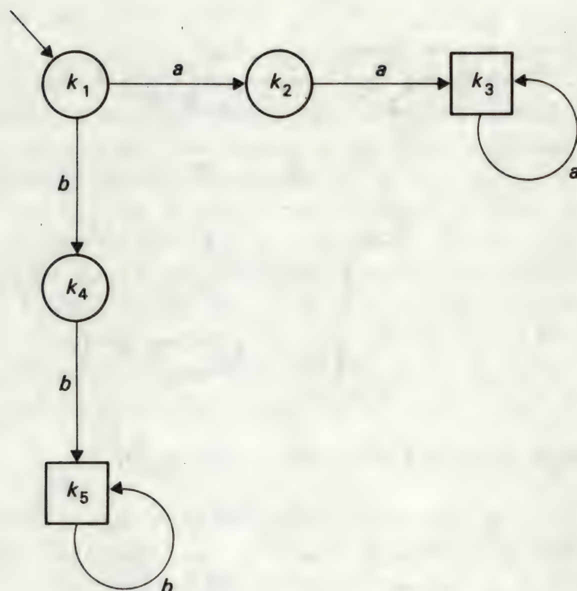
$$k_1 \rightarrow ak_2|bk_4$$
$$k_2 \rightarrow ak_3|a$$
$$k_3 \rightarrow ak_3|a$$
$$k_4 \rightarrow bk_5|b$$
$$k_5 \rightarrow bk_5|b$$

Bewijs nu formeel dat $x \in L(G)$ desd als $x \in T(M) \backslash \{\varepsilon\}$.

(iii) $\Rightarrow$ (i): Deze constructie is in feite de reden waarom we ons onderzoek naar eindige automaten begonnen. Als $L = L(G)$ gegeneerd wordt door $G = (N, T, P, S)$ dan wordt L geaccepteerd door een NFSA $= (N \cup \{\#\}, T, t, S, \{\#\})$. Hierbij is $\#$ een nieuw symbool $\notin N$ en t wordt volgens de volgende regels gedefinieerd. Als $A \rightarrow a$ een produktieregel in P is dan bevat $t(A, a)$ het symbool $\#$ en als $A \rightarrow aB$ een produktieregel in P is dan bevat $t(A, a)$ B. Alle symbolen uit $t(A, a)$ worden volgens deze regels gegeneerd.

Omdat er voor elke reguliere taal L een DFSA, M, bestaat met $T(M) = L$ beschikken we over een efficiënte methode om te testen of een willekeurige string x al dan niet tot L behoort. We construeren eenvoudig de machine M en controleren of er een pad met label x bestaat vanuit de begintoestand naar een eindtoestand. Omdat we eisen dat M deterministisch is, weten we dat er hoogstens één pad is met die eigenschap. We kunnen desgewenst ook een dummy-toestand gebruiken; in dat geval is er altijd precies één pad vanuit de begintoestand voor iedere $x \in T^*$ en we hoeven alleen maar te controleren of dit ons naar een eindtoestand brengt.

Nu we hebben aangetoond dat de reguliere talen juist die talen zijn die door DFSA's of NFSA's worden geaccepteerd, kunnen we met dit resultaat nog een paar interessante stellingen bewijzen.



Figuur 3.5

Stelling 3.4

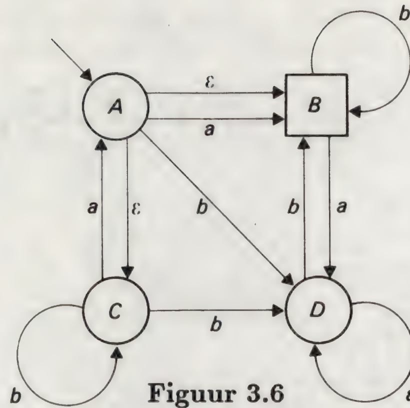
Als L een reguliere taal in T^* is, dan geldt dit ook voor het complement van L, $\overline{L} = T^* \backslash L$.

Bewijs:
Zij $M = (K, T, t, k_1, F)$ een DFSA die L accepteert. Door zo nodig een dummytoestand in te voeren kunnen we ervoor zorgen dat de overgangsfunctie t totaal is. Als $x \in L$, dan is er in de digraaf die M voorstelt een pad met label x vanuit de begintoestand k_1 naar een eindtoestand. Als $x \notin L$ dan moet het pad vanuit k_1 met label x leiden naar een toestand die geen eindtoestand is. $\overline{L}$ wordt dus geaccepteerd door $\overline{M} = (K, T, t, k_1, K \backslash F)$ en dus is ook $\overline{L}$ een reguliere taal.

Stelling 3.5

Als L_1 en L_2 reguliere talen zijn dan geldt dit ook voor hun doorsnede $L_1 \cap L_2$.

Bewijs:
De complementen $\overline{L}_1$ en $\overline{L}_2$ zijn volgens stelling 3.4 beide reguliere talen. Volgens stelling 3.1b geldt dit dus ook voor hun vereniging $\overline{L}_1 \cup \overline{L}_2$. We passen nu opnieuw stelling 3.4 toe en hieruit volgt het te bewijzen, omdat immers $L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}$.



Figuur 3.6

3.3 Eindige automaten met ε-stappen

Als we de definitie van een nondeterministische eindige automaat, (NFSA) zo wijzigen dat er geen input nodig is om van de ene toestand naar de andere te gaan, dan spreken we van een automaat met ε-stappen. Formeler: een NFSA $M = (K, T, t, k_1, F)$, heeft ε-stappen als t in plaats van als een functie $K \times T \rightarrow 2^K$, is gedefinieerd als een functie $K \times (T \cup \{\varepsilon\}) \rightarrow 2^K$. In figuur 3.6 is een voorbeeld getekend van zo'n NFSA.

Als $M = (K, T, t, k_1, F)$ een NFSA is met ε-stappen en $k, k' \in K$ zijn zodanig dat $k' \in t(k, \varepsilon)$ dan noemen we k' *ε-bereikbaar* vanuit k en we schrijven $k \underset{\varepsilon}{-} k'$. In zo'n geval kan M dus van toestand k in toestand k' overgaan zonder dat hiervoor enige input nodig is. Zij nu $\overset{*}{\underset{\varepsilon}{-}}$ de reflexieve transitieve afsluiting van $\overline{\varepsilon}$. In dat geval geldt $k \overset{*}{\underset{\varepsilon}{-}} k'$ desd als $k = k'$ of als er een pad van k naar k' met lengte ≥ 1 bestaat, waarvan elke kant met ε gelabeld is. Als $k \in K$ dan geven we de verzameling van toestanden die vanuit k bereikbaar zijn zonder verdere input weer door $R(k)$. Er geldt dus $R(k) = \{k' | k \overset{*}{\underset{\varepsilon}{-}} k'\}$. We breiden de definitie uit tot: als $K' \subset K$ dan is

$$R(K') = \bigcup_{k \in K'} R(k).$$

In ons voorbeeld geldt $R(A) = \{A, B, C\}$, $R(B) = \{B\}$, $R(C) = \{C\}$ en $R(D) = \{D\}$. Dus $R(\{A, B\}) = R(A) \cup R(B) = \{A, B, C\}$, enzovoort.

We moeten nu eerst de overgangsfunctie t van onze NFSA uitbreiden met ε-stappen tot een functie $\hat{t} : K \times (T \cup \{\varepsilon\}) \rightarrow 2^K$, waarvoor geldt dat als $k \in K$, dan is $\hat{t}(k, a)$ de verzameling toestanden die we kunnen bereiken vanuit toestand k na input a. Wegens de ε-stappen is $\hat{t}$ niet eenvoudig gelijk aan t. Als we in het voorbeeld in toestand A zijn dan geldt $t(A, a) = \{B\}$, maar ook D, A zelf en C zijn bereikbaar met alleen input a. Om in D te komen ga je

eerst naar B zonder input en dan naar D met input a. Om naar A te komen ga je naar C zonder input en dan keer je terug naar A met input a. C kan dan weer zonder input worden bereikt. We definiëren dus $\hat{t} : K \times (T \cup \{\varepsilon\}) \to 2^K$ als

$$\hat{t}(k, \varepsilon) = R(k)$$

en

$$\hat{t}(k, a) = \bigcup_{k' \in R(k)} R(t(k', a)), \quad \text{voorالle } k \in K, a \in T.$$

In het voorbeeld krijgen we

$$\hat{t}(k, a) = R(A) = \{A, B, C\},$$

$$\begin{aligned}\hat{t}(A, a) &= \bigcup_{k' \in \{A,B,C\}} R(t(k', a)) \\ &= R(\{B\}) \cup R(\{D\}) \cup R(\{A\}) \\ &= \{A, B, C, D\},\end{aligned}$$

$$\begin{aligned}\hat{t}(A, b) &= \bigcup_{k' \in \{A,B,C\}} R(t(k', b)) \\ &= R(\{D\}) \cup R(\{B\}) \cup R(\{C, D\}) \\ &= \{B, C, D\},\end{aligned}$$

enzovoort.
We breiden de functie t nu uit tot $2^K \times (T \cup \{\varepsilon\}) \to 2^K$ door voor $K' \subset K$ en $a \in T$ te definiëren

$$\hat{t}(K', \varepsilon) = R(K'),$$

en

$$\hat{t}(K', a) = \bigcup_{k \in K'} \hat{t}(k, a).$$

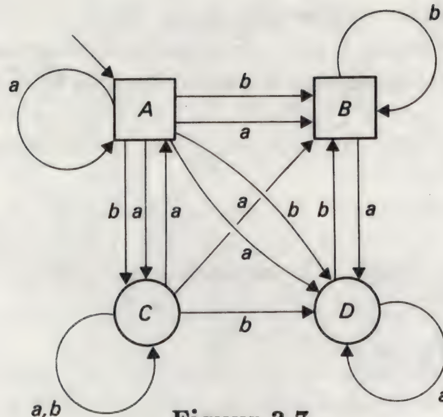
Vervolgens definiëren we $\hat{t} : 2^K \times T^* \to 2^K$ door te stellen

$$\hat{t}(K', ax) = \hat{t}(\hat{t}(K', a), x), \quad \text{voorالle } K' \subset K, a \in T, x \in T^*.$$

De verzameling strings die worden geaccepteerd door de NFSA met ε-stappen $M = (K, T, t, k_1, F)$ definiëren we nu formeel als

$$T(M) = \{x \in T^* | \hat{t}(\{k_1\}, x) \cap F \neq \emptyset\}.$$

Als M zo'n automaat is, dan kunnen we uit M de NFSA $M' = (K, T, t', k_1, F')$ construeren met $t'(k, a) = \hat{t}(k, a)$ en $F' = F$ als $R(k_1) \cap F = \emptyset$ en anders kiezen we $F' = F \cup \{k_1\}$. M' heeft geen ε-stappen en het is niet moeilijk aan te tonen dat $T(M) = T(M')$. Dit leidt tot



Figuur 3.7

Stelling 3.6

Als L wordt geaccepteerd door een NFSA met ε-stappen, dan is L een reguliere taal.

De gelijkwaardige NFSA zonder ε-stappen die we uit ons voorbeeld kunnen construeren is getekend in figuur 3.7.

3.4 Oefeningen

1. Laat L de verzameling strings in $\{0,1\}^*$ voorstellen zodanig dat elke 0 als directe rechter buurman een 1 heeft.
   (a) Construeer een reguliere grammatica die L genereert.
   (b) Construeer een *deterministische* eindige automaat die L accepteert.
2. Zij L een reguliere verzameling. Stel $\text{Pref}(L) = \{x|x$ is een prefix van een string in $L\}$. Bewijs dat $\text{Pref}(L)$ ook regulier is. (Hint: gebruik het feit dat $L = T(M)$ voor een of andere DFSA, M.)
3. Schrijf de formele bewijzen van stelling 3.1b uit.
4. Zij $M = (\{k_1, k_2, k_3\}, \{a, b\}, t, k_1, \{k_3\})$ een NFSA met $t(k_1, a) = \{k_2, k_3\}, t(k_2, a) = \{k_1, k_2\}, t(k_3, a) = \{k_1, k_3\}, t(k_1, b) = \{k_1\}, t(k_2, b) = \emptyset, t(k_3, b) = \{k_1, k_2\}$. Construeer een deterministische automaat die $T(M)$ accepteert.
5. Construeer een DFSA die equivalent is met de NFSA in figuur 3.6.
6. Een *homomorfisme* $\theta : T_1^* \to T_2^*$ is een totale functie met $\theta(\varepsilon) = \varepsilon$ en $\theta(xy) = \theta(x)\theta(y)$ voor alle $x, y \in T_2^*$. Als $T_1 = \{a, b\}, T_2 = \{b, c\}$ en $\theta(a) = bc, \theta(b) = bb$, bepaal dan $\theta(ab)$ en $\theta(ba)$. Laat zien dat als L een reguliere taal is en θ is een willekeurig homomorfisme, dat dan $\theta(L)$ ook een reguliere taal is.

7. Geef het formele bewijs van stelling 3.6.
8. Gebruik stelling 3.6 voor een alternatief bewijs van stelling 3.1a.
9. Beschrijf een eindige automaat die elk woord accepteert dat begint met *'on'* en eindigt met *'k'*. Gebruik deze FSA om een Pascal-programma te schrijven dat Nederlandse tekst inleest en telt hoeveel van dergelijke woorden erin voorkomen.
10. Als iedere produktieregel in een grammatica $G = (N, T, P, S)$ ofwel van de vorm $A \rightarrow Ba$, ofwel van de vorm $A \rightarrow a$ is, met $A, B \in N$ en $a \in T$, laat dan zien dat er een NFSA bestaat die $L(G)$ accepteert. (Hint: de produktie $A \rightarrow Ba$ komt overeen met één kant in een graaf met label a vanuit een knoop B naar een knoop A.)
11. Als iedere produktieregel in een grammatica $G = (N, T, P, S)$ ofwel van de vorm $A \rightarrow xB$, ofwel van de vorm $A \rightarrow x$ is, met $A, B \in N$ en $x \in T^*$, dan heet de grammatica *rechtslineair*. Omgekeerd, als alle regels de vorm $A \rightarrow Bx$ of $A \rightarrow x$ hebben, dan heet de grammatica *linkslineair*. Toon aan dat $L \in T^*$ wordt gegenereerd door een rechtslineaire grammatica dan en slechts dan als L regulier is en dan en

Hoofdstuk 4

Reguliere talen II

In een tekening moeten geen onnodige lijnen staan
en een machine moet geen onnodige onderdelen hebben.
—WILLIAM STRUNCK JR.
The Elements of Style

4.1 Reguliere expressies

Met behulp van reguliere expressies kunnen reguliere talen kort en bondig worden beschreven. De regels die *reguliere expressies over* T definiëren kunnen als volgt worden samengevat:

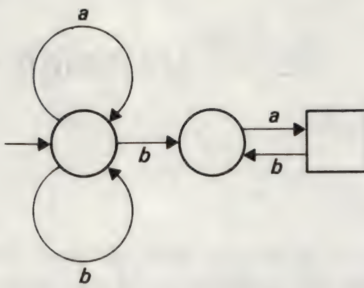
(i) 0 stelt de lege verzameling voor en 1 stelt $\{\varepsilon\}$ voor;
(ii) als $L = \{x_1, x_2, \ldots, x_n\}$ een eindige verzameling strings is met $x_i \in T^*$ voor $i = 1, 2, ..., n$, dan wordt L weergegeven door $(x_1 + x_2 + \ldots + x_n)$;
(iii) als r_1 een reguliere expressie is voor een reguliere taal L_1 en r_2 stelt een reguliere taal L_2 voor dan geldt

$$(r_1 + r_2) \text{ is } L_1 \cup L_2,$$
$$(r_1 \cdot r_2) \text{ is } L_1 L_2,$$
$$(r_1^k) \text{ is } L_1^k \ (k \geq 0)$$
$$(r_i^*) \text{ is } L_i^*;$$

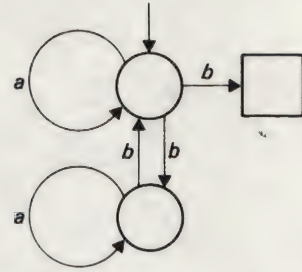
(iv) alleen de expressies gedefinieerd volgens (i), (ii) en (iii) zijn reguliere expressies.

Haakjes kunnen in reguliere expressies worden weggelaten als men uitgaat van de regel dat unaire operatoren de hoogste prioriteit hebben, gevolgd door $\cdot$ en ten slotte $+$. Net als in de gewone algebra wordt voor $(r_1 \cdot r_2)$ meestal $(r_1 r_2)$ geschreven.

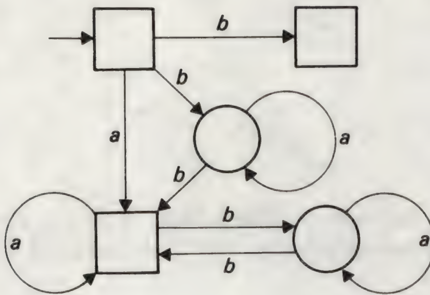
Elke eindige verzameling strings is een reguliere taal en reguliere talen zijn gesloten onder vereniging, verzamelingconcatenatie en Kleene afsluiting. Hieruit volgt dat iedere reguliere expressie over T een reguliere taal in T^* beschrijft. Bijvoorbeeld



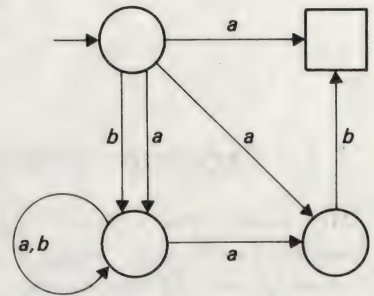
(a) accepteert $(a+b)^*ba(ba)^*$



(b) accepteert $(a+ba^*b)^*b$



(c) accepteert $(a+ba^*b)^*+b$
$= 1+(a+ba^*b)(a+ba^*b)^*+b$



(d) accepteert $(b^*+a^*)^*ab+a$
$= (b+a)^*ab+a$
$= (b+a)(b+a)^*ab+ab+a$

Figuur 4.1

$$(a+b)^*ba(ba)^* \text{ beschrijft } \{a,b\}^*\{ba\}\{ba\}^* = \{a,b\}^*\{ba\}^+$$

en

$$(a+ba^*b)^*+b \text{ beschrijft } (\{a\}\cup\{b\}\{a\}^*\{b\})^*\cup\{b\}.$$

Meestal is het vrij gemakkelijk een nondeterministische eindige automaat (NFSA) te construeren die de taal accepteert die wordt beschreven door een gegeven reguliere expressie. Het is daarbij ook vrij gemakkelijk om vergissingen te maken! Een paar technieken die je bij de constructie kunt gebruiken, worden geïllustreerd in figuur 4.1a-d. In de voorbeelden gebruiken we steeds R om de taal voor te stellen die hoort bij een reguliere expressie R.

U hebt misschien al gezien dat we bij twee van de voorbeelden de expressie wat anders hebben opgeschreven om de constructie wat te vereenvoudigen. Hadden we dat niet gedaan dan was de kans groot geweest dat we in moeilijkheden waren geraakt. Een algebra van reguliere expressies wordt beschreven

door de identiteiten in de regels (a) tot en met (p) en de beweringen (q) en (r) in de hierna volgende stelling 4.1. Met behulp van deze regels kan worden bewezen dat elk paar reguliere expressies dat dezelfde reguliere verzameling beschrijft, equivalent is. Dit bewijs is echter lang niet eenvoudig en later in dit hoofdstuk zullen we zien dat er een betere manier bestaat om de equivalentie van reguliere expressies te bewijzen.

Stelling 4.1

Als Q, R en S willekeurige reguliere expressies over T zijn dan geldt

(a) $Q + Q = Q, Q + 0 = Q$,
(b) $Q + R = R + Q$,
(c) $(Q + R) + S = Q + (R + S) = Q + R + S$,
(d) $(QR)S = Q(RS) = QRS$,
(e) $Q1 = 1Q = Q$ en $Q0 = 0Q = 0$,
(f) $(R + S)Q = RQ + SQ$,
(g) $Q(R + S) = QR + QS$,
(h) $Q^*Q^* = Q^*$,
(i) $(Q^*)^* = Q^*$,
(j) $QQ^* = Q^*Q$,
(k) $Q^* = 1 + Q + Q^2 + \ldots + Q^{k+1}Q^*$ voor alle $k \geq 0$,
(l) $1^* = 1$ en $0^* = 1$,
(m) $(Q^* + R^*)^*(Q^*R^*)^* = (Q + R)^*$,
(n) $(RQ)^*R = R(QR)^*$,
(o) $(Q^*R)^*Q^* = (Q + R)^*$,
(p) $(Q^*R)^* = (Q + R)^*R + 1$ en $(QR^*)^* = Q(Q + R)^* + 1$,
(q) $Q = R^*S$ impliceert $Q = RQ + S$,
(r) als $1 \notin R$ dan volgt uit $Q = RQ + S$ dat $Q = R^*S$.

Bewijs:
De meeste beweringen liggen voor de hand als men voor $+$ vereniging leest en voor $\cdot$ verzamelingconcatenatie. We zullen alleen de twee implicaties bewijzen om een idee te geven van de bewijsmethode

(q) $$\begin{aligned} Q = R^*S &= (RR^* + 1)S && \text{[op grond van (b) en (k)]} \\ &= RR^*S + S && \text{[op grond van (e) en (f)]} \\ &= RQ + S. \end{aligned}$$

(r) $Q = RQ + S$ impliceert Q
$$\begin{aligned} &= R(RQ + S) + S \\ &= R^2Q + RS + S \\ &= R^2(RQ + S) + RS + S \\ &= R^3Q + R^2S + RS + S \\ &= \ldots \\ &= R^{k+1}Q + (R^k + \ldots + R + 1)S \text{ voor alle } k \geq 0. \end{aligned}$$

Voor iedere $x \in Q$ is er een $k \geq 0$ zodat $|x| \leq k$. Omdat $1 \notin R$ geldt $x \notin R^{k+1}Q$ en dus moet $x \in (R^k + \ldots + R + 1)S$, en dus is $x \in R^*S$. Uit $x \in Q$ volgt dus dat $x \in R^*S$. Omgekeerd als $x \in R^*S$, dan is $x \in (R^k + \ldots + R + 1)S$ voor een $k \geq 0$. Uit $x \in R^*S$ volgt dus $x \in Q$. Dus moet $Q = R^*S$.

Elke reguliere expressie beschrijft een reguliere taal zoals we zagen. Het omgekeerde is ook waar. Dit wordt bewezen in de volgende stelling.

Stelling 4.2: De stelling van Kleene.

Elke reguliere taal in T^* kan worden beschreven door een reguliere expressie over T.

Bewijs:
Stel L is een reguliere expressie. Dan geldt $L = T(M)$ voor een of andere deterministische eindige automaat (DFSA), $M = (K, T, t, k_1, F)$. Stel dat $K = \{k_1, k_2, \ldots, k_n\}$. Als nu $F = \{k_{\lambda 1}, k_{\lambda 2}, \ldots, k_{\lambda m}\}$ dan is $L = L_1 \cup L_2 \cup \ldots L_m$. Hierbij is L_i een taal die wordt geaccepteerd door $M = (K, T, t, k_i, \{k_\lambda\})$. Vereniging kunnen we als $+$ weergeven in reguliere expressies, dus het enige dat we moeten bewijzen is dat elke L_i kan worden weergegeven door een reguliere expressie. Dat wil zeggen dat we moeten aantonen dat de verzameling strings die wordt geaccepteerd door een willekeurige DFSA met een begintoestand en een eindtoestand, kan worden voorgesteld door middel van een reguliere expressie.

Beschouw nu $M = (K, T, t, k_1, F)$ met een zodanige labelling van de toestanden dat $K = \{k_1, k_2, \ldots, k_n\}$ en $F = \{k_n\}$. Laat verder T_{ij}^l $(1 \leq, j \leq n, 0 \leq l \leq n)$ de taal voorstellen die bestaat uit alle strings x met $t(k_i, x) = k_j$ en ook $t(k_i, y) \in \{k_m : m \leq l\}$ voor alle geschikte prefixen y van x. (Elke string $y \in T^+$ met $x = yv$ voor een $v \in T^+$ is een geschikt prefix van $x \in T^*$.) We bewijzen door middel van inductie naar l dat T_{ij}^l altijd door een reguliere expressie kan worden voorgesteld. Als $i \neq j$ dan is $T_{ij}^0 = 0$, tenzij er één of meer kanten van knoop k_i naar knoop k_j lopen. In dit laatste geval is $T_{ij}^0 = a_1 + a_2 + \ldots + a_r$, waarbij $a_1, a_2, \ldots, a_r$ de labels bij de kanten voorstellen. Als $i = j$ dan is $T_{ij}^0 = 1$, tenzij er één of meer kanten van knoop k_i naar knoop k_i terugkeren. Dan is $T_{ij}^0 = 1 + a_1 + a_2 + \ldots + a_r$. Als $l = 0$ is T_{ij}^l dus een reguliere expressie. Met behulp van de gelijkheid $T_{ij}^l = T_{ij}^{l-1} + T_{il}^{l-1}(T_{ll}^{l-1})^* T_{lj}^{l-1}$ die geldt voor alle $1 \leq l \leq n$ kunnen we nu eenvoudig met inductie aantonen dat T_{ij}^l steeds een reguliere expressie is. Omdat $T(M) = T_{1n}^n$ is hiermee de stelling bewezen.

Mocht deze redenering nog niet duidelijk zijn, stelt u zich dan de gelabelde digraaf M voor en een willekeurige string $x \in T_{ij}^l$. Er loopt een pad met label x van knoop k_i naar knoop k_j via knopen die als labels elementen uit $\{k_1, k_2, \ldots, k_l\}$ hebben. Het pad loopt ofwel niet via knoop k_l; in dat geval is $x \in T_{ij}^{l-1}$, ofwel het loopt wel via knoop k_l en deze knoop wordt bijvoorbeeld $m \geq 1$ keer bezocht. We kunnen nu het pad van knoop k_i naar knoop k_j partitioneren in stap (0) vanuit knoop k_i naar knoop k_l bij het eerste bezoek, stap (1) van knoop k_l terug naar knoop k_l bij het tweede bezoek, ... stap (m-1)

van knoop k_l terug naar knoop k_l bij het m-de bezoek en stap (m) van knoop k_l naar knoop k_j. We mogen dus schrijven $x = x_0x_1 \dots x_m$, waarbij x_i het label bij het i-de gedeelte van het pad voorstelt. Nu is $x_0 \in T_{il}^{l-1}, x_1, \dots x_{m-1} \in T_{ll}^{l-1}$ en $X_m \in T_{lj}^{l-1}$. Dus is $x \in T_{il}^{l-1}(T_{ll}^{l-1})^*T_{lj}^{l-1}$. Het is nu gemakkelijk in te zien dat elke string in deze verzameling een element van T_{ij}^l is en hieruit volgt de gelijkheid.

4.2 Minimalisatie

Bij de constructie van een deterministische eindige automaat die een of andere taal L accepteert, is het prettig als we zo weinig mogelijk toestanden moeten onderscheiden. Zo'n DFSA heet een minimale DFSA en we zullen hier laten zien hoe je die kunt construeren. We zullen daarbij bewijzen dat deze minimale DFSA uniek is.

Uit hoofdstuk 1 zult u zich herinneren dat een equivalentierelatie op A, A onderverdeelt in een aantal disjuncte equivalentieklassen. De equivalentieklasse die $a \in A$ bevat geven we aan door $\bar{a}$. De *index* van de equivalentierelatie komt overeen met het aantal van deze equivalentieklassen.

We richten nu onze aandacht op equivalentierelaties op T^*. Zo'n equivalentierelatie R heet *rechtsinvariant* als

xRy impliceert dat $xzRyz$ voor alle $z \in T^*$.

Stel L is een taal in T^* en definieer de *relatie geassocieerd met* L, R_L door

xR_Ly desd als, voor alle $z \in T^*$, $xz \in L$ juist als $yz \in L$.

R_L is een equivalentierelatie want R_L is reflexief, symmetrisch en transitief.

Als $M = (K, T, t, k_1, F)$ een DFSA is dan kunnen we ook een *relatie* R_M *geassocieerd met* M definiëren. Twee strings $x, y \in T^*$ hebben relatie R_M dan en slechts dan als zij beide vanuit de begintoestand naar dezelfde toestand leiden, dus

xR_My desd als $t(k_1, x) = t(k_1, y)$.

Ook dit is een equivalentierelatie en ook deze relatie is rechtsinvariant. We beperken ons vooralsnog tot DFSAs met totale overgangsfuncties. In dat geval verdeelt R_M T^* in een aantal disjuncte equivalentieklassen. Iedere equivalentieklasse bestaat uit strings die naar een bepaalde toestand leiden vanuit de begintoestand. Een equivalentieklasse heeft dus een natuurlijk verband met een toestand in K. Omdat er $\#(K)$ toestanden zijn, zijn er ook $\#(K)$ equivalentieklassen en dus heeft R_M een eindige index.

Omdat R_M rechtsinvariant is volgt uit xR_My dat xzR_Myz voor alle $z \in T^*$. Stel dat de taal L door M wordt geaccepteerd, dus $L = T(M)$ dan geldt $xzR_Myz \Rightarrow t(k_1, xz) = t(k_1, yz) \Rightarrow t(k_1, xz) \in F$ desd $t(k_1, yz) \in F \Rightarrow xz \in L$

desd als $yz \in L$. Hiermee hebben we aangetoond dat xR_My, XR_Ly impliceert. Dit betekent dat elke equivalentieklasse gedefinieerd door R_M geheel binnen een equivalentieklasse gedefinieerd door R_L moet liggen. R_M partitioneert T^* in een eindig aantal equivalentieklassen en dus heeft ook R_L een eindige index. Iedere equivalentieklasse bevat een of meer equivalentieklassen van R_M en dus kunnen we met elke equivalentieklasse van R_L een deelverzameling van toestanden van M verbinden. Deze deelverzamelingen verbonden met verschillende equivalentieklassen van R_L zijn onderling disjunct. Als k en k' twee toestanden in K zijn, verbonden met dezelfde equivalentieklasse van R_L, dan geldt $t(k, z) \in F$ voor alle $z \in T^*$, dan en slechts dan als $t(k', z) \in F$. We noemen in dit geval k en k' *ononderscheidbaar*.

Bovenstaande redenering geldt voor iedere DFSA met een totale functie die L accepteert en dus moet zo'n DFSA minstens n toestanden hebben, waarbij n de index van R_L is. Als $M = (K, T, t, k_1, F)$ en als $M' = (K', T, t', k'_1, F')$ twee DFSA's zijn met n toestanden die beide L accepteren, dan kan precies één toestand van elke machine worden geassocieerd met iedere equivalentieklasse van R_L. Dit leidt tot een bijectie $\theta : K \rightarrow K'$, waarvan men eenvoudig kan aantonen dat $\theta(k) = \theta(k') \Rightarrow \theta(t(k, a)) = \theta(t(k', a)), \forall a \in T$. Wegens het bestaan van deze bijectie geldt dat, behoudens de namen van de toestanden, elke DFSA met een totale overgangsfunctie die L accepteert en precies n toestanden heeft, uniek is. Nu moeten we nog aantonen dat er minstens één zo'n automaat bestaat.

Stel dat $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ de equivalentieklassen zijn die door R_L zijn gedefinieerd. We kunnen dan een een DFSA, M_L construeren met inputalfabet T, met toestanden $\{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$, begintoestand $\bar{\varepsilon}$ en eindtoestandenverzameling F, waarbij $\bar{x} \in F$ desd als $x \in L$. We definiëren de overgangsfunctie door $t(\bar{x}, a) = \overline{xa}$ te stellen. Uit de rechtsinvariantie van R_L volgt dat xR_Ly impliceert dan xaR_Lya en dus is deze definitie consistent. Nu is $x \in T(M)$ desd als $t(\bar{\varepsilon}, \bar{x}) = \bar{x} \in F$ desd als $x \in L$ en dus geldt $T(M) = L$.

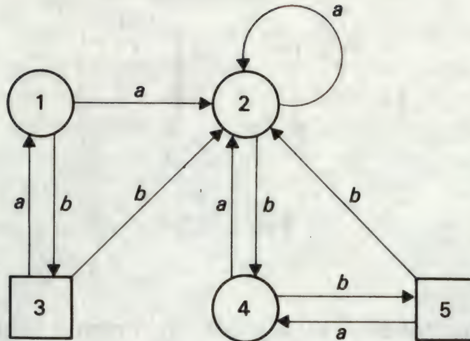
Hiermee hebben we de volgende stelling bewezen.

Stelling 4.3: Stelling van Myhill-Nerode.

Bij elke reguliere taal L bestaat een deterministische eindige automaat met een totale overgangsfunctie die L accepteert en waarvan het aantal toestanden gelijk is aan de index van R_L, de relatie die met L is geassocieerd.

Iedere toestand in een DFSA die wordt geconstrueerd met behulp van de stelling van Myhill-Nerode, die niet op een pad ligt dat loopt van de begintoestand naar een eindtoestand kan geheel worden weggelaten samen met alle verbindingen met die toestand. De DFSA die dan overblijft noemen we de minimale DFSA, M_L die L accepteert.

Zij $M = (K, T, t, k_1, F)$ een DFSA met een totale overgangsfunctie die de taal L accepteert. We beschrijven nu een algoritme dat de minimale machine M_L direct uit L berekent. We moeten daartoe de toestanden van M in disjuncte deelverzamelingen van onderling ononderscheidbare toestanden verdelen. We



Figuur 4.2

definiëren als volgt een rij relaties $D_0, D_1, \ldots$ op de toestandsverzameling K: k is onderscheidbaar van k' door een string met lengte 0, notatie kD_0k', dan en slechts dan als ofwel $k \in F$ en $k' \notin F$, ofwel $k \notin F$ en $k' \in F$. Voor $i > 0$ geldt kD_ik', k is onderscheidbaar van k' door een string met lengte van hoogstens i desd als $kD_{i-1}k'$ of als er een $a \in T$ bestaat zodanig dat $t(k,a)D_{i-1}t(k',a)$. Een toestand k is onderscheidbaar van een toestand k', notatie kDk' desd als er een $i \geq 0$ bestaat met kD_ik'. Met inductie toont men eenvoudig aan dat kD_ik' desd als er een string x met lengte $\leq i$ bestaat zodanig dat ofwel $t(k,x) \in F$ en $t(k',x) \notin F$ of $t(k,x) \notin F$ en $t(k',x) \in F$. Om D te berekenen, berekenen we nu de rij $D_0, D_1, D_2, \ldots$. Zodra $D_r = D_{r+1}$ weten we dat we klaar zijn en dat $D_r = D$. Als $m = \#(K)$ dan bestaan er $m^2 - m$ paren (k_i, k_j) met $i \neq j$. In het slechtste geval verschilt iedere D_{i+1} van d_i in precies twee van die paren en omdat we weten dat D_0 niet leeg is volgt dat $D = D_r$ voor $r < (m^2 - m)/2$.

Als voorbeeld voor deze constructie gebruiken we de in figuur 4.2 getekende DFSA. Elke relatie D_i geven we weer door een 5×5 Boole'se matrix, waarvan het element met index j, k de waarde T (Waar) heeft dan en slechts dan als toestand j door D_i gerelateerd is aan toestand k. De relatie D_i is altijd symmetrisch en de bewering kD_ik geldt voor geen enkele k; we hoeven dus alleen het deel van de matrix boven de hoofddiagonaal weer te geven. In tabel 4.1 staat de rij matrices die overeenkomt met $D_0, D_1 = D_2$. Er geldt $1D_12$, omdat $t(1,b)D_0t(2,b)$ en $2D_14$ omdat $t(2,b)D_0t(4,b)$. De toestanden 1 en 4 en de toestanden 3 en 5 zijn dus ononderscheidbaar.

Als de ononderscheidbare toestanden in een DFSA, M eenmaal zijn bepaald dan kan M_L gemakkelijk worden geconstrueerd. De toestanden van M_L bestaan uit verzamelingen onderling ononderscheidbare toestanden. Stel dat $K' \subset K$ zo'n verzameling voorstelt, dan definiëren we de overgangsfunctie t_L van M_L als $t_L(K', a) = K''$. K'' is de verzameling ononderscheidbare toestanden die $t(k,a)$ bevat voor elke $k \in K'$. De begintoestand van M_L wordt gevormd door de verzameling ononderscheidbare toestanden die de begintoestand van

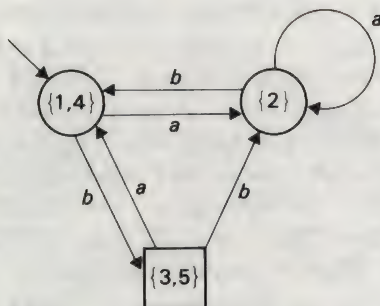
D_0	D_1
FTFT	TTFT
.TFT	.TTT
.TF	.TF
.T	.T
.	

Tabel 4.1

M bevat. De eindtoestanden van M_L worden gevormd door de verzamelingen ononderscheidbare toestanden die alleen eindtoestanden van M bevatten. Als we een en ander op ons voorbeeld toepassen dan krijgen we de minimale DFSA van figuur 4.3. In dit geval ligt elke knoop op een pad van begintoestand naar één der eindtoestanden, dus er hoeven geen knopen te worden verwijderd.

Als we een minimale DFSA beschrijven dan labelen we de knooppunten meestal niet. De ongelabelde DFSA is de unieke minimale DFSA die de taal L accepteert.

Heeft de oorspronkelijke DFSA een partiële overgangsfunctie dan voeren we een 'dummy'-toestand Δ in en construeren we een equivalente DFSA met een totale functie. Als we vervolgens bovenstaande constructie toepassen op deze DFSA dan kan de verzameling ononderscheidbare toestanden vanuit Δ niet liggen op een pad van een begintoestand naar een eindtoestand en dus komt niet voor in de minmale DFSA.



Figuur 4.3

4.3 Algoritmen voor reguliere grammatica's

Stel M is een willekeurige reguliere taal waarvoor we een minimale DFSA M_L hebben geconstrueerd met $T(ML) = L$. Veronderstel dat M_L als toestandsverzameling $K = \{k_1, k_2, \ldots, k_n\}$ heeft met k_1 als begintoestand en beschouw $z \in L$ met lengte van minimaal n. Om door M_L geaccepteerd te worden moet $t(k_1, z) \in F$. Omdat $|z| \geq n$ moet er een toestand $k \in K$ zijn met $z = wxy$ als $|x| \geq 1$ en $t(k_1, w) = k, t(k, x) = k$ en $t(k, y) \in F$. Maar dan is ook $t(k_1, wx^iy) \in F$ voor $i \geq 0$ en dus geldt $wx^iy \in L$ voor $i \geq 0$. Hiermee hebben we de volgende stelling bewezen.

Stelling 4.4

Als L een reguliere taal is en de string $z \in L$ heeft een lengte groter gelijk het aantal toestanden in M_L dan geldt $z = wxy$ met $|x| \geq 1$ en $wx^iy \in L$ voor $i \geq 0$.

Gevolg van deze stelling is dat uit het feit dat L niet leeg is volgt dat er een string $z \in L$ met lengte $< n$ bestaat, waarbij n het aantal toestanden in M_L is. Een eenvoudig algoritme[1] om te testen of $L(G) = \emptyset$ of niet luidt als volgt: construeer de minimale DFSA M_L voor $L = L(G)$. Als M_L n toestanden heeft dan is $L = \emptyset$ desd als M_L geen enkele string in T^* accepteert met lengte $< n$. Er bestaat slechts een eindig aantal van dergelijke strings en dus kan dit in een eindige tijd worden nagegaan. Hieruit volgen de volgende stellingen.

Stelling 4.5

Het leegheidsprobleem voor reguliere grammatica's is oplosbaar. Dat wil zeggen: gegeven een reguliere grammatica G bestaat er een algoritme waarmee kan worden vastgesteld of $L(G)$ leeg is of niet.

Verder geldt:

Stelling 4.6

(a) Het equivalentieprobleem voor reguliere grammatica's is oplosbaar. Gegeven twee grammatica's G_1 en G_2 bestaat er een algoritme waarmee kan worden vastgesteld of $L(G1) = L(G2)$ of niet.
(b) Het eindigheidsprobleem voor reguliere grammatica's is oplosbaar: gegeven een reguliere grammatica G, bestaat er een algoritme dat bepaalt of $L(G)$ al of niet eindig is.

Bewijs:

(a) $L(G_1) = L(G_2)$ desd als de minimale DFSA's geconstrueerd uit G_1 en G_2, behoudens de naamgeving van hun toestanden, identiek zijn.

[1] Een algoritme dat een gegeven probleem π oplost kan worden gedefinieerd als een procedure die met elke verschijningsvorm van π als input altijd na eindige tijd de gewenste oplossing produceert.

(b) Ook hier is het bewijs gebaseerd op de minimale DFSA $M_L = (K,T,t,k_1,F)$, geconstrueerd voor de acceptatie van $L = L(G)$. Als M_L n toestanden heeft dan weten we op grond van stelling 4.4 dat L dan en slechts dan oneindig is als er een $z \in L$ met lengte van minimaal n bestaat. Stel dergelijke strings bestaan en stel dat w van die strings de kleinste lengte heeft, d.w.z. als $|w| \geq n$ en $|z| \geq n$, dan is $|w| \leq |z|$. We bewijzen nu dat $|w| < 2n$. Want stel dat dit niet zo was dan schrijven we $w = w_1w_2w_3$ met $1 \leq |w_2| \leq n$ en met $t(k_1, w_1) = k, t(k, w_2) = k, t(k, w_3) \in F$ voor een $k \in K$. Dus is $t(k_1, w_1w_3) \in F$ en dus is $w_1w_3 \in L$. Echter $|w_1w_3| = |w| - |w_2| \geq 2n - n = n$ en $|w_1w_3| < |w|$ en dit leidt tot een tegenspraak. Ons algoritme bestaat dus uit niets anders dan uit het construeren van de minimale DFSA die $L = L(G)$ accepteert. Als deze DFSA n toestanden heeft dan gaan we na of zij een string z accepteert met $n \leq |z| < 2n$. Het aantal van dergelijke strings is oneindig evenals L dan en slechts dan als minstens één zo'n string wordt geaccepteerd.

Het is de lezer misschien al opgevallen dat de bewijzen van de stellingen 4.5 en 4.6b niet afhangen van de voorwaarde dat de DFSA minimaal is. Elke DFSA met n toestanden kan worden gebruikt, maar bij een minimale DFSA is dit aantal zo klein als mogelijk is en dus moeten dan ook zo min mogelijks strings worden getest. We vatten nu het voorgande samen in de laatste stelling van dit hoofdstuk.

Stelling 4.7

Het leegheids-, equivalentie- en eindigheidsprobleem is oplosbaar voor deterministische (en nondeterministische) eindige automaten.

4.4 Oefeningen

1. Beschrijf deterministische eindige automaten die de volgende expressies accepteren.
   (a) $a(ba + b)^* + b$,
   (b) $(ab + b^*)^*ba + b$,
   (c) $((b^*a)^*ab^*)^*$.
2. Bepaal een reguliere expressie die $T(M)$ beschrijft als M de automaat uit oefening 3.4 is.
3. Als $R_1, \ldots, R_n$ reguliere expressies over T zijn en $f(R_1, \ldots, R_n)$ is een functie die alleen de operatoren $+, \cdot$ en $*$ gebruikt dan geldt
   (a) $f(R_1, \ldots, R_n) + (R_1 + \ldots + R_n)^* = (R_1 + \ldots + R_n)^*$;
   (b) $(f(R_1^*, \ldots, R_n^*))^* = (R_1 + \ldots + R_n)^*$.

   Bewijs dit.

4. Gebruik stelling 4.4 om aan te tonen dat de volgende verzamelingen geen reguliere talen voorstellen:
   (a) $\{a^n b^n | n \geq 0\}$,
   (b) $\{xx^r | x \in T^*\}$.
5. Bepaal een deterministische eindige automaat die de taal L beschreven door de expressie $(ab)^* + (a)(ba + a)^*$ accepteert. Construeer vervolgens de minimale DFSA die L accepteert. Gebruik deze laatste automaat om een automaat te construeren die $\overline{L} = a, b^* \backslash L$ accepteert. Formuleer nu een reguliere expressie die L beschrijft.
6. Zij R_M de relatie geassocieerd met een DFSA, $M = (K, T, t, k_1, F)$ zodanig dat $T(M) = L$. Stel verder dat $\bar{x}$ en $\bar{y}$ twee verschillende equivalentieklassen van R_M zijn. Bewijs dat als $t(k_1, x) = k$ en $t(k_1, y) = k'$ dat dan xR_Ly geldt dan en slechts dan als k en k' onderling ononderscheidbaar zijn.
7. Bekijk nog eens de oplossingen van opgave 4.1 en ga na of en zo ja welke DFSA's die u construeerde minimaal zijn.

Hoofdstuk 5

Contextvrije Talen

Om de vorm te tonen die het scheen te verbergen.
—SIR WALTER SCOTT
The Lord of the Isles

In hoofdstuk 2 zagen we dat er twee equivalente definities van contextvrije grammatica's bestaan. Ten eerste kunnen we een contextvrije grammatica (CFG) definiëren als een frasestructuurgrammatica (N, T, P, S) waarbij iedere produktieregel de vorm $A \to \alpha$ heeft met $A \in N$ en $\alpha \in (N \cup T)^*$. Op grond van deze definitie is een willekeurig aantal ε-produkties in de grammatica toegelaten. Bij de tweede definitie, als $\varepsilon \notin L(G)$ kunnen we de produktieregels in een CFG beperken tot $A \to \alpha, A \in N$ en $\alpha \in (N \cup T)^+$, waarbij dus ε-produkties vermeden worden. Als we ook $\varepsilon \in L(G)$ willen toelaten dan kiezen we voor maar één ε-produktie, $S \to \varepsilon$ en eisen dat S niet voorkomt als een deelstring in het rechterlid van enig andere produktieregel. Meestal zullen we voor deze tweede definitiewijze kiezen omdat deze formele bewijzen vaak wat eenvoudiger maakt.

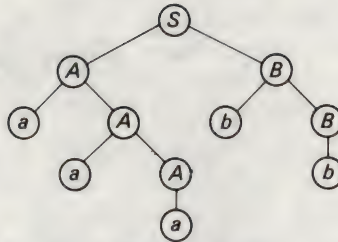
Als een taal L wordt gegenereerd door een CFG dan heet L een contextvrije taal (Engels: Context-free Language of CFL). Elke reguliere grammatica is contextvrij en dus is elke reguliere taal een contextvrije taal. Er bestaan echter wel degelijk CFL's die niet regulier zijn. Een eenvoudig voorbeeld is de niet-reguliere taal $\{a^n b^n | n \geq 1\}$ die wordt gegenereerd door de CFG met produktieregels

$$S \to aSb|ab$$

Zij $G = (N, T, P, S)$ een willekeurige CFG en stel $L(G) \neq \emptyset$. Bij elke $w \in T^+$ die we door middel van G genereerden behoort een afleidingsboom. De diepte van een boom definieerden we eerder als het langste pad vanuit de wortel naar een blaadje. Als G_1 de volgende produktieregels heeft

$$S \to AB$$
$$A \to aA|a$$
$$B \to bB|b$$

dan heeft de afleidingsboom voor $a^3b^2 \in L(G)$ die in figuur 5.1 is weergegeven diepte vier.



Figuur 5.1

Heeft de afleidingsboom voor een CFG, $G = (N, T, P, S)$ diepte n dan moet er een pad bestaan met knooppunten $A_1, A_2, \ldots, A_n, a$. Hierbij is $S = A_1, A_2, \ldots, A_n \in N$ en $a \in T$. Is $n > \#(N)$ dan moeten twee of meer knopen op dit pad hetzelfde label hebben, bijvoorbeeld $A_i = A_j$ en $i < j$. We kunnen nu een nieuwe afleidingsboom construeren door de boom met als wortel A_i te vervangen door de boom met als wortel A_j. Als we deze redenering herhaald toepassen dan volgt hieruit dat als er een afleidingsboom voor $w \in L(G)$ bestaat met diepte $> \#(N)$ dan moet er ook een afleidingsboom met diepte $\leq \#(N)$ bestaan voor een of andere $w' \in L(G)$. Hieruit volgt

Stelling 5.1

Het leegheidsprobleem voor contextvrije grammatica's is oplosbaar. Dit wil zeggen dat er voor elke CFG, $G = (N, T, P, S)$ een algoritme bestaat waarmee kan worden vastgesteld of al dan niet geldt dat $L(G) = \emptyset$.

Bewijs:
Het algoritme bestaat er eenvoudig uit dat we testen of $S \to \varepsilon$ een produktieregel in P is. Als dit zo is dan is $L(G) \neq \emptyset$ omdat immers $\varepsilon \in L(G)$. Als dit niet het geval is dan construeren we alle mogelijke afleidingsbomen, *maar slechts tot diepte* $\#(N)$. Hiervan bestaat een eindig aantal en geen van deze bomen komt overeen met een afleiding van een terminale string dan en slechts dan als $L(G) = \emptyset$.

Een produktie $A \to \alpha$ in een CFG $G = (N, T, P, S)$ heet *relevant* dan en slechts dan als er een afleiding voor een $x \in L(G)$ bestaat die die produktieregel gebruikt, d.w.z. desd als $S \overset{*}{\Rightarrow} \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \alpha \alpha_2 \overset{*}{\Rightarrow} x$ voor een $x \in L(G)$. Als een produktieregel in G niet relevant is heet deze *irrelevant* en we zouden hem graag willen verwijderen. Voor iedere $A \in N$ kunnen we een nieuwe grammatica $G_A = (N, T, P, A)$ construeren. Als $L(G_A)$ de lege verzameling is dan kunnen we geen terminale string vanuit A genereren met behulp van produkties uit P. Dit betekent dat we uit geen enkele zinsvorm uit G die A bevat een terminale string kunnen afleiden. $L(G)$ blijft dus onveranderd als we alle produktieregels uit G verwijderen die A ofwel in het linker- ofwel in het rechterlid hebben staan. Er kunnen nu nog enkele irrelevante produktieregels

over zijn in de grammatica omdat er regels $A \in N$ kunnen voorkomen die terminale strings kunnen genereren, maar die nooit in een zinsvorm kunnen voorkomen. Op overeenkomstige wijze als in stelling 5.1 kunnen we bewijzen dat als A in een zinsvorm kan voorkomen dat er dan een zinsvorm bestaat die A bevat en die kan worden afgeleid met behulp van een partiele afleidingsboom met diepte $< \#(N)$. (De boom heet een *partiële* afleidingsboom omdat zijn blaadjes geen terminalen zijn.) Door al dergelijke bomen te genereren kunnen we nagaan of er een $A \in N$ bestaat die niet in een zinsvorm kan voorkomen. Als dit het geval is dan kan elke produktieregel waar A in voorkomt als zijnde irrelevant worden verwijderd. Van nu af zullen we dan ook veronderstellen dan alle produktieregels in een contextvrije grammatica relevant zijn.

5.1 De normaalvorm van Chomsky

De vorm van produktieregels in CFG's is nogal vrij en deze vrijheid maakt bewijzen van eigenschappen van CFG's erg moeilijk. Gelukkig kunnen we aan de regels vrij strenge beperkingen opleggen terwijl de zeggingskracht van de grammatica toch behouden blijft.

Stelling 5.2: De Normaalvorm van Chomsky

Elke ε-vrije CFL kan worden gegenereerd door een CFG in de Normaalvorm van Chomsky.

Deze CFG heeft de volgende produktieregels

$$A \to BC, A, B, C \in N$$

of

$$A \to a, A \in N, a \in T$$

Bewijs:
Als $L \subset T^*$ een ε-vrije CFL is dan mogen we aannemen dat L wordt gegenereerd door een CFG, $G = (N, T, P, S)$ zonder ε-produkties. Een equivalente CFG in de normaalvorm van Chomsky (met dus alleen produktieregels van de vorm die in de stelling wordt omschreven), kunnen we uit G construeren met behulp van het volgende algoritme.

Stap 1: (Vervang alle produkties van de vorm $A \to B$ met $A, B \in N$; we noemen dit soort produktieregels *eenheidsproduktieregels*.) Geef voor elke $A \in N$ de verzameling van eenheidsproduktieregels met A in het linkerlid aan door $U(A)$ en geef de verzameling van niet-eenheidsproduktieregels met A in het rechterlid aan door $N(A)$. Vervang voor iedere $A \in N$ waarvoor $U(A)$ niet leeg is $U(A)$ door $\{A \to \alpha | A \overset{+}{\underset{G}{\Rightarrow}} B$ en $B \to \alpha$ in $N(B)\}$.

Stap 2: (Vervang alle produkties met als rechterleden strings met lengte > 1 en met terminalen als substrings. Dergelijke produktieregels heten *secundaire produktieregels*.) Introduceer voor iedere $a \in T$ die in het rechterlid voorkomt

van een secundaire produktieregel een nieuwe nonterminaal A_a samen met een nieuwe produktieregel $A_a \rightarrow a$. Elke secundaire produktieregel van de vorm $A \rightarrow X_1X_2 \dots X_m, X_i \in N \cup T$ wordt nu vervangen door een produktieregel $A \rightarrow Y_1Y_2 \dots Y_m$ met $Y_i = X_i$ als $X_i \in N$; anders is $X_i = a$ voor een $a \in T$ en $Y_i = X_a$. De vermeerderde verzameling van nonterminalen geven we aan door N'.

Stap 3: (Vervang alle produktieregels met als rechterleden strings van meer dan twee nonterminalen. Deze regels heten *tertiaire produktieregels.*) Elke tertiaire produktie van de vorm $A \rightarrow B_1B_2 \dots B_m, m > 2, B_1, B_2, \dots, B_m \in N'$ wordt vervangen door produkties $A \rightarrow B_1B_1', B_1' \rightarrow B_2B_2', \dots, B_{m-1}' \rightarrow B_{m-1}B_m$. $B_1', \dots, B_{m-1}'$ zijn nonterminalen die niet in andere produktieregels mogen voorkomen.

Een grammatica die met behulp van de bovenstaande stappen uit G wordt geconstrueerd is equivalent met G en is in de normaalvorm van Chomsky. Beschouw bijwijze van voorbeeld de CFG G_2 met als produktieregels

$$S \rightarrow A|ABA$$
$$A \rightarrow aA|a|B$$
$$B \rightarrow bB|b$$

Stap 1: $U(S)$ omvat $S \rightarrow A, U(A)$ omvat $A \rightarrow B$ en $U(B)$ is leeg. Omdat $S \stackrel{+}{\Rightarrow} A$ en $S \stackrel{+}{\Rightarrow} B$ vervangen we $S \rightarrow A$ door $S \rightarrow aA|a|bB|b$ en omdat $A \stackrel{+}{\Rightarrow} B$ vervangen we $A \rightarrow B$ door $A \rightarrow bB|b$. De equivalente grammatica heeft dan de volgende produktieregels

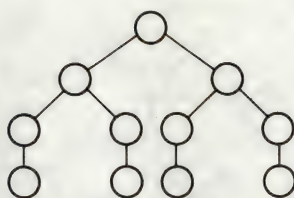
$$S \rightarrow aA|a|bB|b|ABA$$
$$A \rightarrow aA|a|bB|b$$
$$B \rightarrow bB|b$$

Stap 2: a en b staan beiden in een rechterlid van een secundaire produktie. We introduceren daarom twee nieuwe nonterminalen A_a en A_b te zamen met de produktieregels $A_a \rightarrow a$ en $A_b \rightarrow b$. We herschrijven de secudaire produktieregels en krijgen zo de volgende serie equivalente produktieregels

$$S \rightarrow A_aA|a|A_bB|b|ABA$$
$$A \rightarrow A_bA|a|A_bB|b$$
$$B \rightarrow A_bB|b$$
$$A_a \rightarrow a$$
$$A_b \rightarrow b$$

Stap 3: De enige tertiaire produktieregel is $S \rightarrow ABA$. Deze wordt vervangen door de volgende twee regels

$$S \rightarrow AB'$$
$$B' \rightarrow BA$$



Figuur 5.2

en zo krijgen we de uiteindelijke contextvrije grammatica in de normaalvorm van Chomsky.

Stel $L \subset T^*$ is een willekeurige contextvrije taal en $L \backslash \{\varepsilon\}$ wordt gegenereerd door een contextvrije grammatica G in de normaalvorm van Chomsky. Als $x \in L(G)$ een afleidingsboom met diepte m heeft dan is eenvoudig aan te tonen dat $|x| \leq 2^{m-1}$. Dit volgt uit het feit dat elke ouder in de boom hoogstens twee kinderen kan hebben, terwijl ouders van blaadjes er maar een hebben. In figuur 5.2 hebben we een afleidingsboom met diepte 3 van maximale omvang getekend.

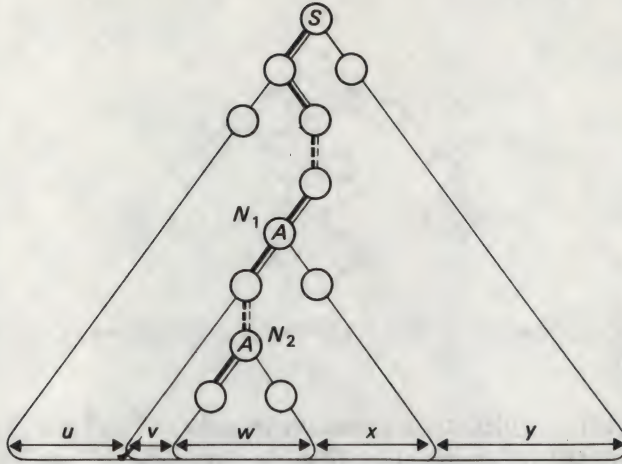
We bewijzen nu de volgende stelling.

Stelling 5.3

Als L een contextvrije taal is dan bestaan er gehele getallen l_1 en l_2 zodanig dat voor elke $z \in L$ met $|z| > l_1, z$ geschreven kan worden als $z = uvwxy$, waarbij

(i) $|vwx| \leq l_2$,
(ii) $vx \neq \varepsilon$, en
(iii) voor elk geheel getal $i > 0$ geldt $uv^iwx^iy \in L$.

Bewijs:
Stel $L \backslash \{\varepsilon\}$ wordt gegenereerd door een grammatica $G = (N, T, P, S)$ in normaalvorm van Chomsky. Zij $n = \#(N)$, $l_1 = 2^{n-1}$ en $l_2 = 2^n$. Stel verder $z \in L$ met $|z| > 2^{n-1}$. Elke afleidingsboom voor z moet nu een langste pad P hebben met lengte groter dan n. Er zitten dan meer dan $n + 1$ knopen in P, waarvan er een terminaal is. Dit betekent dat er twee knopen N_1 en N_2 in P moeten zitten, die beide zijn gelabeld met hetzelfde nonterminale symbool. Stel N_1 is de knoop het dichtst bij de wortel. Er kunnen meer dergelijke paren in P zitten, maar we kiezen het paar met N_1 op de grootste diepte in de boom. In dat geval vormen N_1 en N_2 het enige paar met identiek label op het pad vanuit N_1 naar het eindpunt. De lengte van dit pad is hoogstens $n + 1$. Stel N_1 en N_2 hebben beide label A. Omdat P het langste pad is in de afleidingsboom, is het pad vanuit N_1 ook het langste pad vanuit de subboom met wortel N_1. De diepte van deze subboom is hoogstens $n + 1$ en moet dus een afleiding representeren van een terminale string vanuit A met een lengte van hoogstens 2^n. Noem deze string z_1. We hebben aangetoond dat $|z_1| \leq 2^n$ en dat $A \overset{*}{\Rightarrow} z_1$. Als T_2 de sub-



Figuur 5.3

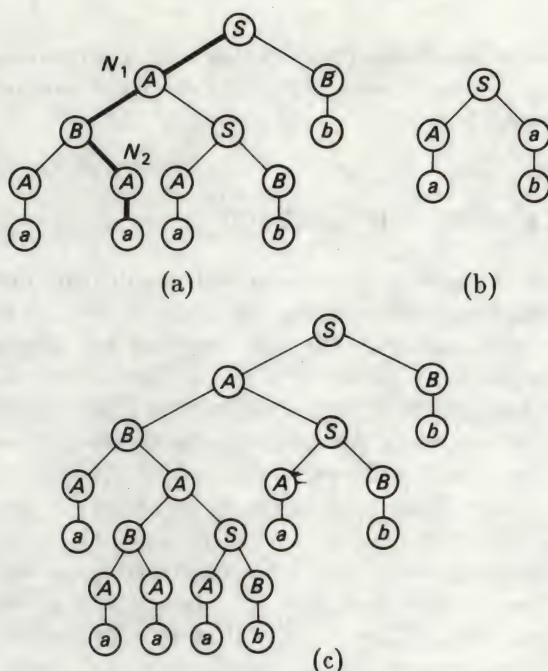
boom met wortel N_2 is en w is de terminale string die uit deze subboom wordt afgeleid dan kunnen we $z_1 = vwx$ schrijven, waarbij v en x niet beide ε kunnen zijn. Immers de eerste produktieregel gebruikt in de afleiding van z_1 moet de vorm $A \rightarrow BC$ hebben voor een $B, C \in N$ en ε-produkties zijn niet toegelaten. Nu is z_1 een substring van z en dus kunnen we schrijven $z = uz_1y = uvwxy$. We hebben hiermee aangetoond dat er een afleiding $S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy$ bestaat met $|vwx| \leq l_2$. Omdat $A \overset{*}{\Rightarrow} vAx \overset{*}{\Rightarrow} vwx$ geldt $A \overset{*}{\Rightarrow} v^iwx^i$ voor iedere $i > 0$. Dus $S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uv^iwx^iy$ is een toegelaten afleiding in G voor iedere $i \geq 0$. In figuur 5.3 is dit nog eens geïllustreerd.

We gebruiken ter illustratie van deze stelling de grammatica G_3 met produktieregels

$$S \rightarrow AS|AB$$
$$A \rightarrow BS|a$$
$$B \rightarrow AA|b$$

In dit geval is $l_1 = 4$; we beschouwen daarom de afleidingsboom voor a^3b^2 in figuur 5.4a. In de boom is een pad P met maximale lengte aangegeven. In dit pad zitten vier nonterminale knopen. Minstens twee moeten dus hetzelfde label hebben; bijvoorbeeld de twee knopen met label A. Noem nu N_1 de knoop A het dichtst bij de wortel en noem de andere N_2. De boom T_1 met N_1 als wortel geeft de afleiding $A \overset{*}{\Rightarrow} a^3b$ en dit is de string z_1 uit de stelling. T_2 is de boom met wortel N_2 en met afleiding $A \Rightarrow a$. Dus $z_1 = vwx$ met $v = a$, $w = a$, $x = ab$ en $z = uvwxy$, waarbij $u = \varepsilon$ en $y = b$. Volgens de stelling geldt voor elke gehele $i \geq 0$ dat $uv^iwx^iy \in L$. In de figuren 5.4b en 5.4.c zijn de afleidingsbomen voor $i = 0$ en $i = 2$ getekend. Figuur 5.4b wordt uit 5.4a geconstrueeerd door T_1 door T_2 te vervangen en 5.4c volgt uit 5.4a door het vervangen van T_2 door T_1.

Stelling 5.3 is vooral bruikbaar als je wilt bewijzen dat een bepaalde taal



Figuur 5.4

niet contextvrij is. Neem bijvoorbeeld eens de taal $L = \{a^p | p \text{ is een priemgetal}\}$. Als L contextvrij was dan zou uit stelling 5.3 volgen dat p is priem $\Rightarrow p + 2ki$ is priem voor een k met $0 < k \leq p$ en alle $i \geq 0$. Dit is niet waar en dus is L niet contextvrij.

Stel G is een willekeurige contextvrije grammatica in de normaalvorm van Chomsky. Als we aannemen dat G n nonterminalen bevat en als $l_1 = 2^{n-1}$ en $l_2 = 2^n$, dan geldt wegens stelling 5.3 dat $L(G)$ oneindig is dan en slechts dan als $L(G)$ een string met lengte $> l_1$ bevat. Als we ervan uitgaan dat $L(G)$ oneindig is, stel dan dat z de kortste string in $L(G)$ is met lengte $> l_1$. Door een tegenspraak af te leiden zullen we bewijzen dat $l_1 < |z| \leq l_1 + l_2$. Immers, stel dat $|z| > l_1 + l_2$ terwijl er geen kortere string in L bestaat met lengte $> l_1$, dan geldt op grond van stelling 5.3 dat $z = uvwxy$ en $uwy \in L$ is een string met lengte $|z| - l_2 > l_1$, maar $|uwy| < |z|$ en dit is de gezochte tegenspraak.

We bewezen dus dat $L(G)$ oneindig is dan en slechts dan als er een $z \in L(G)$ bestaat met $l_1 < |z| \leq l_1 + l_2$. Er bestaat maar een eindig aantal strings $\in T^*$ die aan deze lengtebeperking voldoen. Door alle zinsvormen van G met lengte $k, l_1 < k \leq l_1 + l_2$ te genereren kunnen we nagaan of er terminale strings in $L(G)$ bij zijn. We hebben zo dus een algoritme verkregen dat test of $L(G)$ eindig is of niet. Met andere woorden we bewezen

Stelling 5.4

Het eindigheidsprobleem is oplosbaar voor contextvrije grammatica's. Dat wil zeggen er bestaat een algoritme om te bepalen of een CFG een eindige of een oneindige taal genereert.

5.2 De normaalvorm van Greibach

Toen we in hoofdstuk 3 reguliere grammatica's behandelden vonden we gelukkig een heel efficiënte methode waarmee de vraag 'is $x \in T^*$ een element van $L(G)$?' kan worden beantwoord. Dit probleem heet het *elementprobleem* (Engels: membership problem). Duidelijk is dat dit probleem ook voor contextvrije grammatica's oplosbaar is; we hoeven immers alleen maar alle zinnen uit G met lengte $|x|$ te genereren. Als geen van die zinnen met een afleiding van x overeenkomt dan is $x \notin L(G)$ en anders is $x \in L(G)$. Dit is geen erg efficiënte manier om het probleem op te lossen en we zullen een groot deel van het vervolg van dit hoofdstuk wijden aan het zoeken naar betere methoden. Bij het compileren van een programmatekst is het construeren van een afleidingsboom van belang. Daarom zijn we ook in de volgende vraag geinteresseerd: 'is $x \in T^*$ een element van $L(G)$, en zo ja, wat is de bijbehorende afleidingsboom?'. Dit probleem noemen we het *afleidingsprobleem*, (Engels: derivation problem). Als we hiervoor een efficiënt algoritme kunnen vinden dan hebben we meteen het elementprobleem opgelost, maar tot nu toe weten we alleen maar dat het probleem inderdaad oplosbaar is.

In de praktijk blijkt het afleidingsprobleem alleen efficiënt oplosbaar als we bepaalde restricties aan de CFG opleggen. Afhankelijk van de aard van de restricties zijn verschillende algoritmen ontwikkeld voor het oplossen van het afleidingsprobleem. Helaas zijn die restricties zodanig dat niet alle contextvrije talen eraan voldoen. In de hoofdstukken 6, 7 en 8 gaan wij hier nader op in. Ter voorbereiding is het van belang dat u leert produktieregels te transformeren zonder dat hierdoor de gegenereerde taal verandert. Ook bij de behandeling van de normaalvorm van Chomsky maakten we hiervan al gebruik.

De eerste transformatietechniek is heel eenvoudig en wordt samengevat in de volgende stelling waarvan het bewijs aan de lezer wordt overgelaten.

Stelling 5.5

Als $A \rightarrow \alpha_1 B \alpha_2$ een produktieregel in een contextvrije grammatica G is en $B \rightarrow \beta_1|\beta_2|\ldots|\beta_k$ zijn alle mogelijke produktieregels met B in het linkerlid dan kan $A \rightarrow \alpha_1 B \alpha_2$ vervangen worden door $A \rightarrow \alpha_1\beta_1\alpha_2|\alpha_1\beta_2\alpha_2|\ldots|\alpha_1\beta_k\alpha_2$ zonder dat de taal $L(G)$ hierdoor verandert.

De volgende stelling ligt misschien niet zo voor de hand en we geven daarom een aanwijzing voor het bewijs. De stelling heeft betrekking op het verwijderen van zogenaamde *links-recursieve* produktieregels, d.w.z. produktieregels van de vorm $A \rightarrow A\alpha, A \in N, \alpha \in (N \cup T)^*$ uit een contextvrije grammatica.

Stelling 5.6

Als in een CFG, $A \to A\alpha_1|A\alpha_2|\dots|A\alpha_m$ alle mogelijke links-recursieve produktieregels zijn met A in het linkerlid en $A \to \beta_1|\beta_2|\dots|\beta_n$ zijn de overige produktieregels met A in het linkerlid, dan kan een equivalente grammatica worden geconstrueerd door een nieuwe nonterminaal A' in te voeren en de genoemde produktieregels te vervangen door

$$A \to \beta_1|\beta_2|\dots|\beta_n|\beta_1 A'|\beta_2 A'|\dots|\beta_n A'$$
$$A \to \alpha_1|\alpha_2|\dots|\alpha_m|\alpha_1 A'|\alpha_2 A'|\dots|\alpha_m A'$$

Aanwijzing voor het bewijs: in beide gevallen is de verzameling strings die uit A kan worden afgeleid met behulp van één of meer van de produktieregels gelijk aan $\{\beta_1, \beta_2, \dots, \beta_n\}\{\alpha_1, \alpha_2, \dots, \alpha_m\}^*$.

Ter illustratie van het gebruik van beide stellingen converteren we de produktieregels van G_3 in produkties van de vorm $A \to a\alpha, A \in N$, $a \in T, \alpha \in (N \cup T)^*$. Als alle produktieregels van een contextvrije grammatica deze vorm hebben dan heeft de grammatica de normaalvorm van Greibach. Het is bij elke ε-vrije CFL L mogelijk een CFG in de normaalvorm van Greibach te construeren die de taal L genereert. De algemene constructiewijze verloopt zoals in het nu volgende voorbeeld.

Eerst geven we andere namen aan de nonterminalen van G_3; we noemen S A_1, we gebruiken A_2 in plaats van A en A_3 voor B. De grammatica heeft dan de volgende produktieregels

$$A_1 \to A_2 A_1|A_2 A_3$$
$$A_2 \to A_3 A_1|a$$
$$A_3 \to A_2 A_2|b$$

Vervolgens zorgen we ervoor dat alle produkties van de vorm $A_i \to A_j\alpha$ voldoen aan $j > i$. In dit voorbeeld voldoet maar één produktieregel hier niet aan, namelijk $A_3 \to A_2 A_2$. Door stelling 5.5 toe te passen, vervangen we deze regel door

$$A_3 \to A_3 A_1 A_2|a A_2$$

Dit is een links-recursieve produktieregel en we gebruiken nu stelling 5.6 om deze te verwijderen. De nieuwe verzameling regels luidt nu

$$A_1 \to A_2 A_1|A_2 A_3$$
$$A_2 \to A_3 A_1|a$$
$$A_3 \to a A_2|b|a A_2 A_3'|b A_3'$$
$$A_3' \to A_1 A_2|A_1 A_2 A_2'.$$

Alle regels met A_3 in het linkerlid hebben nu de vereiste vorm. $A_2 \to A_3 A_1$ kan met behulp van stelling 5.5 worden herschreven en dit geldt ook voor

$A_1 \rightarrow A_2A_1|A_2A_3$. Nu hebben alle regels met A_1, A_2 of A_3 in het linkerlid de gewenste vorm. We krijgen

$$\begin{aligned}
A_1 \rightarrow\ & aA_2A_1A_1|bA_1A_1|aA_2A_3'A_1A_1|bA_3'A_1A_1|aA_1 \\
& |aA_2A_1A_3|bA_1A_3|aA_2A_3'A_1A_3|bA_3'A_1A_3|aA_3 \\
A_2 \rightarrow\ & aA_2A_1|bA_1|aA_2A_3'A_1|bA_3'A_1|a \\
A_3 \rightarrow\ & aA_2|baA_2A_3'|bA_3' \\
A_3' \rightarrow\ & A_1A_2|A_1A_2A_3'
\end{aligned}$$

Tenslotte kunnen we nog de produktieregels met A_3' in het linkerlid herschrijven door stelling 5.5 toe te passen. We krijgen dan

$$\begin{aligned}
A_3' \rightarrow\ & aA_2A_1A_1A_2|bA_1A_1A_2|aA_2A_3'A_1A_1A_2|bA_3'A_1A_1A_2|aA_1A_2 \\
& |aA_2A_1A_3A_2|bA_1A_3A_2|aA_2A_3'A_1A_3A_2|bA_3'A_1A_3A_2|aA_3A_2 \\
& |aA_2A_1A_1A_2A_3'|bA_1A_1A_2A_3'|aA_2A_3'A_1A_1A_2A_3' \\
& |bA_3'A_1A_1A_2A_3'|aA_1A_2A_3'|aA_2A_1A_3A_2A_3'|bA_1A_3A_2A_3' \\
& |aA_2A_3'A_1A_3A_2A_3'|bA_3'A_1A_3A_2A_3'|aA_3A_2A_3'
\end{aligned}$$

Deze grammatica in de normaalvorm van Greibach heeft dus negenendertig produktieregels! We moeten nu nog nagaan of de constructie die we in dit voorbeeld gebruikten ook in het algemeen werkt. We beschrijven eerst de stappen die we uitvoerden en vervolgens bewijzen we dat zij altijd tot een grammatica in de normaalvorm van Greibach leiden.

Stap 1: Zij G een willekeurige ε-vrije grammatica. Converteer G in een equivalente grammatica G' in de normaalvorm van Chomsky.

Stap 2: Noem de nonterminale symbolen in G', $A_1, A_2, \ldots, A_m (m \geq 1)$, waarbij A_1 het beginsymbool voorstelt. Zij verder $N = \{A_1, A_2, \ldots, A_m\}$.

Stap 3: Pas het volgende algoritme toe om alle produktieregels in G' met A_i in het linkerlid te converteren naar ofwel de vorm $A_i \rightarrow a\alpha, a \in T$ ofwel de vorm $A_i \rightarrow A_j\alpha$ met $j > i$.

```
begin
var i: integer;
i := 0;
while i <> m do
        begin
        i := i + 1;
                while er een produktie A_i → A_j, j < i bestaat do
                    vervang A_j volgens stelling 5.5;
                if er links-recursieve produkties bestaan met A_i
                    in het linkerlid
                then voer een nieuwe nonterminaal A'_i in
                    en gebruik stelling 5.6 om ze door een equivalente
                    reeks produktieregels te vervangen
        end
end.
```

Stap 4: Stel N' is de verzameling van nieuwe nonterminale symbolen die we in stap 3 introduceerden. Na stap 3 hebben dan alle regels van de grammatica de vorm $A_i \to A_j\alpha, j > i$ en $\alpha \in (N \cup N' \cup T)^*$ of $A'_i \to X\alpha, X \in (N \cup T), \alpha \in (N \cup N' \cup T)^*$. Gebruik nu het volgende algoritme om alle regels met de eerste vorm te vervangen:

```
begin
var i: integer;
i := m;
while i <> 1 do
        begin
        i := i − 1;
        while er een regel van de vorm A_i → A_j, j > i bestaat do
            vervang A_j volgens stelling 5.5
        end
end.
```

Stap 5: Verwijder nu regels van de vorm $A'_i \to A_j\alpha$ door A_j met behulp van stelling 5.5 te vervangen. De grammatica heeft nu de gewenste vorm.

Duidelijk is dat na stap 2 de grammatica $L(G)$ zal genereren als we A_1 als startsymbool gebruiken. Om aan te tonen dat stap 3 inderdaad zijn doel bereikt, bewijzen we met inductie dat het algoritme na i iteraties alle produkties met $A_k, k \leq i$ in het linkerlid heeft aangepast. Na 0 iteraties geldt de bewering in ieder geval. Stel nu dat de bewering waar is voor $i < n$ en beschouw de n-de iteratie. Als er produktieregels van de vorm $A_n \to A_j\alpha$ zijn met $j < n$ dan gebruiken we stelling 5.5 om A_j te vervangen door de rechterleden van produkties met A_j in het linkerlid. Omdat $j < n$ geldt wegens de inductieveronderstelling dat deze rechterleden ofwel met een terminaal symbool, ofwel met $A_k, k > j$ beginnen. Elke regel van het eerste type is acceptabel en alleen als er regels van het tweede type voorkomen met $k < n$ dan wordt de binnenste lus van het algoritme gebruikt. Maximaal zijn er $n - 1$ iteraties van de binnenste lus mogelijk en hierna hebben alle regels met A_n in het linkerlid rechterleden die beginnen met ofwel een terminaal symbool, ofwel met een nonterminaal symbool A_k met $k \geq n$. Alle links-recursieve regels met A_n in het linkerlid zijn nu vervangen en dus geldt de inductieveronderstelling ook voor $i = n$. Hiermee is bewezen dat het algoritme uit stap 3 zijn doel bereikt. Het stopt als $i = m$ en dat hebben alle produktieregels de gewenste vorm. Als we nagaan hoe stelling 5.5 in stap 3 wordt gebruikt dan is in te zien dat de rechterleden van alle produktieregels waarin een nonterminaal symbool werd ingevoerd in het linkerlid moeten beginnen met een element uit $N \cup T$. De bewering in het begin van stap 4 kan dus worden bevestigd. Met name geldt dat het rechterlid van een regel met A_m in het linkerlid moet beginnen met een terminaal symbool. Elke regel met A_{m-1} in het linkerlid begint of met een terminaal symbool, of met A_m. Voorkomens van produktieregels van het type $A_{m-1} \to A_m\alpha$ kunnen worden verwijderd door stelling 5.5 weer te gebruiken. Nu hebben alle regels met A_m of A_{m-1} in het linkerlid een rechterlid dat met een terminaal symbool

begint. Dit proces herhalen we totdat alle regels met $A_m, A_{m-1}, \ldots, A_2, A_1$ als linkerleden, rechterleden hebben die met een terminaal symbool beginnen. Weer met volledige inductie kunnen we formeel bewijzen dat deze situatie na n iteraties van de lus in stap 4 is bereikt. Ook stap 4 bereikt dus het gewenste resultaat omdat het algoritme na $m-1$ iteraties eindigt. De juistheid van stap 5 is triviaal en dus geldt de gehele constructie. Hiermee hebben wij de volgende stelling bewezen.

Stelling 5.7: De normaalvorm van Greibach

Bij iedere ε-vrije contextvrije taal L bestaat er een contextvrije grammatica G in de normaalvorm van Greibach, zodanig dat $L = L(G)$.

5.3 Contextvrije talen als oplossingen van vergelijkingen

We bekijken opnieuw de contextvrije grammatica met produktieregels

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA|a \\
B &\rightarrow bB|b.
\end{aligned}
$$

Aan elk nonterminaal symbool in deze grammatica kunnen we een verzameling terminale strings verbinden die eruit kunnen worden afgeleid. Aan A wijzen we de verzameling $\{a^n|N \geq 1\}$ toe, aan B de verzameling $\{b^n|n \geq 1\}$ en aan S de verzameling $\{a^m b^n|m, n \geq 1\}$. In deze paragraaf gebruiken we het nonterminale symbool zelf om de bijbehorende verzameling aan te geven en we schrijven dus $A = \{a^n|n \geq 1\}, B = \{b^n|n \geq 1\}$ en $S = \{a^m b^n|m, n \geq 1\}$. Op grond van de produktieregels van de grammatica weten we dat de verzamelingen S, A en B aan het volgende stelsel vergelijkingen moeten voldoen

$$
\begin{aligned}
S &= AB \\
A &= \{a\}A \cup \{a\} \\
B &= \{b\}B \cup \{b\}
\end{aligned}
$$

Dit stelsel heeft een oneindig aantal mogelijke oplossingen. Hoewel bijvoorbeeld een oplossing van

$$A = A \cup \{a\}$$

$\{a\}$ is, is elke verzameling die a bevat ook een oplossing. Deze andere oplossingen liggen niet direct voor de hand als oplossing voor de vergelijking. We zullen in het hierna volgende een unieke 'kleinste' oplossing toekennen aan elk systeem van vergelijkingen.

Ga uit van het volgende stelsel

$$X_1 = f_1(X_1, X_2, \ldots, X_n)$$
$$X_2 = f_2(X_1, X_2, \ldots, X_n)$$
$$\ldots$$
$$\ldots$$
$$\ldots$$
$$X_n = f_n(X_1, X_2, \ldots, X_n)$$

Ieder van de functies $f_i(X_1, X_2, \ldots, X_n)$ wordt geconstrueerd uit eindige verzamelingen strings in T^* en tussen de variabelen $X_1, X_2, \ldots, X_n$ worden alleen vereniging en concatenatie als operaties gebruikt. Een dergelijk stelsel noemen we een stelsel van verzamelingvergelijkingen over T^*. We schrijven $\boldsymbol{X}$ voor het n-tupel $(X_1, X_2, \ldots, X_n)$ en we schrijven het stelsel als

$$\boldsymbol{X} = f(\boldsymbol{X}).$$

Elk n-tupel verzamelingen in T^*, $\boldsymbol{S} = (S_1, S_2, \ldots, S_n)$ dat voldoet aan $\boldsymbol{S} = f(\boldsymbol{S})$ is een oplossing voor $\boldsymbol{X} = f(\boldsymbol{X})$. Als $\boldsymbol{T} = (T_1, T_2, \ldots, T_n)$, dan schrijven we $\boldsymbol{S} \subset \boldsymbol{T}$ desd als $S_1 \subset T_1, S_2 \subset T_2, \ldots, S_n \subset T_n$. De volgende stelling zegt dat elk stelsel verzamelingvergelijkingen over T^* een unieke 'kleinste' oplossing heeft.

Stelling 5.8

Het stelsel $\boldsymbol{X} = f(\boldsymbol{X})$ heeft een oplossing

$$\boldsymbol{S} = \bigcup_{i=1}^{\infty} f^i(\emptyset)$$

en als T enige andere oplossing is dan geldt $S \subset T$.

Bewijs (schets):
$\emptyset$ stelt het n-tupel $(\emptyset, \emptyset, \ldots, \emptyset)$ voor en dus geldt per definitie dat $\emptyset \subset f(\emptyset)$. Op grond van oefening 1.11 kan gemakkelijk worden bewezen dat uit $\boldsymbol{A} \subset \boldsymbol{B}$ volgt dat $f(\boldsymbol{A}) \subset f(\boldsymbol{B})$. Dus geldt $f(\emptyset) \subset f(f(\emptyset)) = f^2(\emptyset)$, enzovoort. De rij $\emptyset \subset f(\emptyset) \subset f^2(\emptyset) \subset \ldots$ is niet-dalend. Als nu

$$\boldsymbol{S} = \bigcup_{i=1}^{\infty} f^i(\emptyset)$$

de limiet van de rij voorstelt dan kan men bewijzen dat $f(\boldsymbol{S}) = \boldsymbol{S}$ en dus is $\boldsymbol{S}$ een oplossing van het stelsel.

Als $\boldsymbol{T}$ een andere oplossing is dan geldt $\boldsymbol{T} = f(\boldsymbol{T})$. Per definitie is $\emptyset \subset \boldsymbol{T}$ en dus is $f(\emptyset) \subset f(\boldsymbol{T}) = \boldsymbol{T}$. Op deze wijze kan men uiteindelijk aantonen dat $f^i(\emptyset) \subset \boldsymbol{T}$ voor alle $i \geq 0$ en dus

$$\bigcup_{i=1}^{\infty} f^i(\emptyset) \subset \boldsymbol{T}, \text{ d.w.z. } \boldsymbol{S} \subset \boldsymbol{T}$$

Bij iedere contextvrije grammatica $G = (N, T, P, S)$ behoort een systeem van verzamelingvergelijkingen over T^* en ook het omgekeerde is het geval. Als $S = (S_1, S_2, \ldots, S_n)$ de unieke kleinste oplossing is van dit stelsel en als de eerste vergelijking overeenkomt met de produktieregels in P met S in het linkerlid dan geldt $L(G) = S_1$. We passen dit eens toe op het volgende stelsel

$$
\begin{aligned}
S &= AB = f_1(S, A, B) \\
A &= \{a\}A \cup \{a\} = f_2(S, A, B) \\
B &= \{b\}B \cup \{b\} = f_3(S, A, B). \\
f(\emptyset) &= (f_1(\emptyset), f_2(\emptyset), f_3(\emptyset)) = (\emptyset, \{a\}, \{b\}), \\
f^2(\emptyset) &= (f_1(\emptyset, \{a\}, \{b\}), f_2(\emptyset, \{a\}, \{b\}), f_3(\emptyset, \{a\}, \{b\})) \\
&= (\{ab\}, \{a, aa\}, \{b, bb\}),
\end{aligned}
$$

en

$$
\begin{aligned}
f^3(\emptyset) &= f(\{ab\}, \{a, aa\}, \{b, bb\}) \\
&= (\{ab, aab, abb, aabb\}, \{a, aa, aaa\}, \{b, bb, bbb\}).
\end{aligned}
$$

Met inductie kan men aantonen dat

$$f^i(\emptyset) = (\{a^m b^n | i > m, n \geq 1\}, \{a^m | i \geq m \geq 1\}, \{b^n | i \geq n \geq 1\})$$

en dus geldt voor de limiet

$$\bigcup_{i=1}^{\infty} f^i(\emptyset) = (\{a^m n^n | m, n \geq 1\}, \{a^m | m \geq 1\}, \{b^n | n \geq 1\}).$$

Hieruit volgt dat $L(G) = \{a^m b^n | m, n \geq 1\}$.

Het formuleren van contextvrije talen als oplossingen van een stelsel van verzamelingvergelijkingen leidt vaak tot een beter begrip van de taal. Ook geldt dat na een eindig aantal stappen in de rij $\emptyset \subset f(\emptyset) \subset \ldots$ alle strings op $L(G)$ van gegeven lengte zijn bepaald. Het elementprobleem kan dus op deze wijze worden opgelost.

5.4 Oefeningen

1. Beschrijf een algoritme dat gegeven een willekeurige contextvrije grammatica G en een geheel getal $k \geq 0$ alle afleidingsbomen voor G met diepte k genereert.
2. Als L een contextvrije taal is bewijs dan dat de Kleene insluiting van L, L^* en de omkering van $L, L^r = \{x^r | x \in L\}$ ook contextvrije talen zijn.
3. Als L_1 en L_2 CFL's zijn dan geldt dit ook voor $L_1 \cup L_2$ en $L_1 L_2$. Bewijs dit.
4. Construeer een contextvrije grammatica in de normaalvorm van Chomsky die rekenkundige expressies over $\{a, b, c\}$ genereert.

5. Welke van de volgende verzamelingen zijn contextvrije talen en welke zijn dat niet?
   (a) $\{a^i b^j c^k | 0 \leq i < j < k\}$,
   (b) $\{a^i b^j c^k | 0 <= j = k\}$,
   (c) $\{a^i b^j c^k | 0 <= j, 0 \leq k \text{ en } i \neq k\}$,
   (d) $\{a^i b^j c^k | 0 <= j, 0 \leq k\}$.
6. Als $L \subset T_1^*$ een CFL is en $\phi : T_1^* \rightarrow T_2^*$ is een homomorfisme dan is ook $\phi(L)$ een CFL. Toon dit aan.
7. Formuleer de formele bewijzen van de stellingen 5.5 en 5.6.
8. Formuleer en bewijs een stelling analoog aan stelling 5.6 die kan worden gebruikt om alle rechts-recursieve produkties uit een CFG te verwijderen. (Dit zijn produktieregels van de vorm $A \rightarrow \alpha A, A \in N, \alpha \in (N \cup T)\alpha)$.
9. Gegeven is een grammatica met als produktieregels

$$A_1 \rightarrow A_2 A_3$$
$$A_2 \rightarrow A_3 A_1 | b$$
$$A_3 \rightarrow A_1 A_2 | a$$

   Converteer deze grammatica naar de normaalvorm van Greibach.
10. Gegeven is een contextvrije grammatica G met produktieregels

$$S \rightarrow AS | BS | a$$
$$A \rightarrow BB | a$$
$$B \rightarrow AA | b$$

   Gevraagd $L(G)$ als een oplossing van een stelsel verzamelingvergelijkingen te formuleren. Gebruik vervolgens dit stelsel om alle strings $x \in L(G)$ met lengte kleiner dan vijf te bepalen.
11. Gebruik uw oplossing van oefening 5.5 om twee contextvrije talen L_1 en L_2 te construeren, zodanig dat hun doorsnede $L_1 \cap L_2$ geen CFL is.
12. Gebruik de oplossing van opgave 5.11 om aan te tonen dat er een CFL, $L \subset T^*$ bestaat met een complement $\overline{L} = T^* \backslash L$ dat geen CFL is.

Hoofdstuk 6

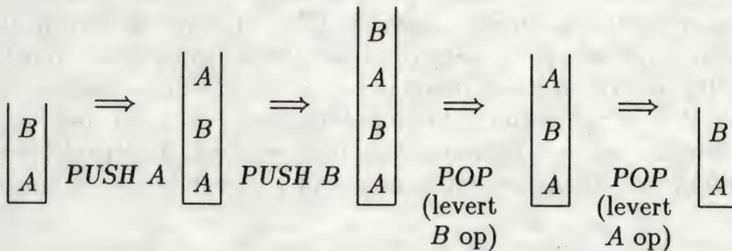
Stapelautomaten

If anyone anything lacks
He'll find it all ready in stacks.

Mist iemand ooit ergens iets
Dan ontstaan er stapels uit het niets.
—SIR WILLIAM SCHWENCK GILBERT
The Sorcerer

De eindige automaat die we in de hoofdstukken 3 en 4 ontmoetten bleek een eenvoudig hanteerbaar model voor de behandeling van reguliere talen. In dit hoofdstuk beschrijven we een soortgelijke automaat als model voor contextvrije talen. Dit is een belangrijke eerste stap in de richting van het ontwerp en de bouw van een praktisch te gebruiken doelmatige ontleder en voor de compilerbouwer is dit een essentieel onderdeel.

De automaat die we zullen gebruiken heeft een geheugen in de vorm van een stapel (Engels: stack). Een *stack* is een lineair opslagsysteem voor gegevens volgens het 'last-in, first-out'-principe, waarmee de meeste lezers met enige informaticakennis wel op de hoogte zullen zijn. Een stack heeft een *top* en een *bodem* en op een stack zijn twee operaties mogelijk PUSH en POP genaamd. PUSH plaatst een nieuw element op de top van de stack en POP neemt een element van de top en levert dit af als resultaat, (zie figuur 6.1). Binnen onze toepassing zullen we de elementen van de stack voorstellen met symbolen uit een alfabet V en veronderstellen we dat de opslagcapaciteit, dus de stackhoogte onbeperkt is. We kunnen dus altijd de PUSH-operatie uitvoeren; een stack



Figuur 6.1

overflow is onmogelijk. Stack underflow is natuurlijk wel mogelijk: bij een POP moet er altijd minstens een element op de stack staan; als we POP op een *lege* stack toepassen dan ontstaat de *underflow* conditie.

Als V het alfabet is van de verzameling symbolen die op de stack kunnen staan, dan kan een stack-inhoud worden weergegeven door een string in V^*. Het eerste symbool van de string geeft de top van de stack weer. We beschrijven dan de elementen van de top naar de bodem van de stack door een string van links naar rechts te doorlopen. Het voorbeeld in figuur 6.1 wordt dan beschreven door $BA \Rightarrow ABA \Rightarrow BABA \Rightarrow ABA \Rightarrow BA$. De lege stack geven we weer door de lege string ε. De stackoperatoren PUSH en POP kunnen we nu formeel definiëren als functies op strings

$$\text{PUSH}(A, X) = AX$$

definieert het toevoegen van symbool $A \in V$ aan een stack met als inhoud de string $X \in V^*$.

$$\text{POP}_1(X) = \begin{cases} \text{niet gedefinieerd} & \text{als } X = \varepsilon \text{ (underflow)}, \\ A & \text{als } X = AY, A \in V, Y \in V^*, \end{cases}$$

definieert het symbool dat wordt opgeleverd door een POP toegepast op een stack met inhoud X.

$$\text{POP}_2(X) = \begin{cases} \text{niet gedefinieerd} & \text{als } X = \varepsilon \text{ (underflow)}, \\ Y & \text{als } X = AY, A \in V, Y \in V^*, \end{cases}$$

definieert de inhoud van de stack die oorspronkelijk inhoud X had, nadat er een POP op is toegepast.

6.1 Nondeterministische stapelautomaten

Een nondeterministische stapelautomaat (Engels: nondeterministic pushdown automaton, afgekort NPDA) is een nondeterministische eindige automaat met een stack waarvan het topelement de overgangsfunctie kan beinvloeden. Bij een nondeterministische eindige automaat (NFSA) wordt de verzameling van mogelijke volgende toestanden bepaald door de huidige toestand en het inputsymbool. Bij een NPDA hangt de verzameling van mogelijke volgende toestanden af van de huidige toestand, het inputsymbool en het symbool dat van de top van de stack wordt gehaald met een POP-operatie. Tijdens de toestandsovergang kan de NPDA een willekeurig eindig aantal symbolen op de stack plaatsen.

Nu de formele definitie: een nondeterministische stapelautomaat (NPDA) is een 7-tupel $M = (K, T, V, p, k_1, A_1, F)$ met

(a) K is een eindige verzameling toestanden,
(b) T is een eindig inputalfabet,

(c) V is een eindig alfabet van stacksymbolen (we schrijven ze meestal als hoofdletters uit het begin van het alfabet, mogelijk met subscript),
(d) p is een totale functie $K \times V \times (T \cup \{\varepsilon\})$ naar eindige deelverzamelingen van $K \times V^*$, Deze functie heet de *stapelfunctie*, (Engels: pushdown function)
(e) $k_1 \in K$ is een tevoren aangewezen *begintoestand*,
(f) $A_1 \in V$ is een tevoren aangewezen *startsymbool* en dit is initieel het enige symbool op de stack,
(g) $F \subset K$ is een verzameling van *eindtoestanden*.

De *configuratie* van een NPDA, $M = (K, T, V, p, k_1, A_1, F)$ kan op ieder tijdstip worden beschreven door een geordend paar uit $K \times V^*$. Het eerste element in dit paar beschrijft de huidige toestand van de automaat en het tweede element beschrijft de huidige stackinhoud. Als een NPDA, M, in configuratie c is dan definiëren we de *overgangsfunctie* t_M zo dat $t_M(c, x)$ de verzameling configuraties voorstelt waarin M kan verkeren na een input $x \in T^*$. We definiëren $t_M : (K \times V^*) \times (T \cup \{\varepsilon\}) \rightarrow 2^{K \times V^*}$ om een stap van de automaat te beschrijven en breiden vervolgens de functie uit.

Als het eenmalig toepassen van de stapelfunctie p veroorzaakt dat M van configuratie $c = (k, AX), k \in K, A \in V, X \in V^*$ overgaat in configuratie $c' = (k', YX), k' \in K, Y, X \in V^*$ gegeven een input $a \in T \cup \{\varepsilon\}$, dan willen we dat $t_M(c, a)$ c' bevat. We definiëren daarom t_M zo dat $t_M((k, AX), a)$ dan en slechts dan (k', YX) bevat als $p(k, A, a)$ (k', Y) bevat.

We spreken af dat $t_M(c, \varepsilon)$ altijd c bevat, maar andere configuraties zitten er alleen in als M zonder input vanuit c naar een andere configuratie kan overgaan. Dit laatste is het geval als $c = (k, AX)$ zodanig is dat $p(k, A, \varepsilon) \neq \emptyset$. We noemen zo'n overgang een *ε-stap*.

Het eerste deel van de definitie van t_M is gereed als we nog afspreken dat er geen overgang mogelijk is als de stack leeg is. Dus

$$t_M((k, \varepsilon), a) = \emptyset \qquad \text{voor alle } k \in K, a \in T$$

en

$$t_M((k, \varepsilon), \varepsilon) = \{(k, \varepsilon)\} \qquad \text{voor alle } k \in K.$$

We breiden nu t_M uit naar de gezochte functie $(K \times V^*) \times T^* \rightarrow 2^{K \times V^*}$. Als $k \in K, X \in V^*, a_1, a_2, \ldots, a \in T \cup \{\varepsilon\}$ dan geldt

$$\begin{aligned}
&t_M((k, X), a_1 a_2 \ldots a_n) \text{ bevat } (k', X'), k' \in K, X' \in V^* \\
&\text{desd als } t_M((k, X), a_1) \text{ bevat } (k_2, X_2), \\
&\qquad t_M((k_2, X_2), a_2) \text{ bevat } (k_3, X_3), \\
&\qquad\qquad \ldots \\
&\qquad\qquad \ldots \\
&\qquad\qquad \ldots \\
&\qquad t_M((k_n, X_n), a_n) \text{ bevat } (k', X'),
\end{aligned}$$

voor enige $k_2, \ldots, k_n, X_2, \ldots, X_n \in V^*$. Merk op dat sommige a_i's ε kunnen zijn als M ε-stappen kan doen.

Nu kunnen we $T(M)$ definiëren: de *verzameling van strings die worden geaccepteerd door de NPDA, M*.

$$T(M) = \{x \in T^* | t_M((k_1, A_1), x) \text{ bevat een element uit } F \times V^*\}.$$

$T(M)$ is dus de verzameling strings die na input kunnen zorgen dat de begintoestand van de automaat uiteindelijk overgaat in een van de eindtoestanden. Tenslotte merken we op dat we het subscript M zullen laten vervallen in de overgangsfunctie t_M als duidelijk is om welke NPDA het gaat.

We construeren nu bij wijze van voorbeeld een NPDA M met $T(M_1) = \{xx^r | x \in \{a, b\}^*\}$. We doen dit door de eerste helft van de string op de stack op te slaan. Als de tweede helft van de string moet worden verwerkt dan controleren we eenvoudig of het ingevoerde symbool overeenkomt met het symbool dat we van de stack halen. Hierbij hebben we wel een probleem: de input wordt element voor element ingevoerd en we kennen de totale lengte dus pas als de hele inputstring is ingevoerd. We kunnen daarom niet weten wanneer we moeten stoppen met het op de stack zetten van ingevoerde symbolen en wanneer we het vergelijkingsproces moeten beginnen. Dit is de reden dat we van een nondeterministische automaat gebruik moeten maken. Zolang we symbolen op de stack zetten blijven we in begintoestand k_1. Op ieder willekeurig moment kunnen we het vergelijkingsproces beginnen door over te gaan naar toestand k_2, maar dan kunnen we niets meer op de stack plaatsen. We stellen nu $M_1 = (\{k_1, k_2, k_3\}, \{a, b\}, \{a, b, \dashv\}, p, k_1, \dashv, \{k_3\})$. Het teken $\dashv$ geeft de bodem van de stack aan. De stapelfunctie definiëren we als volgt (we zullen in het vervolg bij de beschrijving van een stapelfunctie de regels die de lege verzameling genereren weglaten)

$$\begin{aligned}
p(k_1, a, a) &= \{(k_1, aa), (k_2, \varepsilon)\}, \\
p(k_1, b, a) &= \{(k_1, ab)\}, \\
p(k_1, a, b) &= \{(k_1, ba)\}, \\
p(k_1, b, b) &= \{(k_1, bb), (k_2, \varepsilon)\}, \\
p(k_1, \dashv, a) &= \{(k_1, a, \dashv)\}, \\
p(k_1, \dashv, b) &= \{(k_1, b, \dashv)\}, \\
p(k_2, a, a) &= \{(k_2, \varepsilon)\}, \\
p(k_2, b, b) &= \{(k_2, \varepsilon)\}, \\
p(k_2, \dashv, \varepsilon) &= \{(k_3, \varepsilon)\}.
\end{aligned}$$

U gelieve na te gaan dat $T(M_1) = \{xx^r | x \in \{a, b\}^*\}$. Als we bijvoorbeeld $t((k_1, \dashv), abba)$ bekijken, dan kan hieruit iedere configuratie in $\{(k_1, abba \dashv), (k_2, \dashv), (k_3, \varepsilon)\}$ volgen en dus is $abba \in T(M_1)$.

Volgens stelling 3.3 wordt iedere reguliere taal $L \subset T^*$ geaccepteerd door een of andere nondeterministische eindige automaat, $M = (K, T, t, k_1, F)$. Uit deze NFSA kunnen we een NDPA, $M' = (K, T, \{\dashv\}, p, k_1, \dashv, F)$ construeren met $T(M') = L$. M' simuleert het gedrag van M door geen rekening te houden met de stackinhoud. Hiertoe definiëren we p als

$p(k, \dashv, a)$ bevat $(k', \dashv)$ desd als $k' \in t(k, a)$.

De stackinhoud blijft dus $\dashv$ tijdens alle stappen die door M' worden uitgevoerd. Duidelijk is dat $x \in T(M')$ desd als $t_{M'}((k_1, \dashv), x) \cap (F \times \{\dashv\}) \neq \emptyset$ desd als $t(k_1, x) \cap F \neq \emptyset$ desd als $x \in T(M)$ desd als $x \in L$.

Uit het bovenstaande volgt dat elke reguliere taal door één of andere nondeterministische stapelautomaat wordt geaccepteerd. Op grond van ons voorbeeld weten we ook dat men een NDPA kan construeren die een niet-reguliere taal accepteert, namelijk $\{xx^r | x \in \{a, b\}^*\}$. NDPA's accepteren dus een strikt grotere klasse talen dan eindige automaten. We zullen zelfs aantonen dat het juist de klasse der contextvrije talen betreft. Misschien vraagt u zich af of het nondeterminisme wel een nodige eigenschap is. Jammer genoeg is dat inderdaad het geval: deterministische en nondeterministische eindige automaten accepteren wel dezelfde klasse van talen, maar dit geldt niet voor stapelautomaten. Men kan aantonen dat nondeterminisme vereist is voor het herkennen van de taal $\{xx^r | x \in \{a, b\}^*\}$. Later in dit hoofdstuk komen we terug op deterministische automaten.

6.2 Het accepteren van contextvrije talen door nondeterministische stapelautomaten

De belangrijkste eigenschap die we in dit hoofdstuk afleiden is het accepteren van CFL's door NDPA's. Voor elke CFL, L bestaat er een NDPA die die taal accepteert en omgekeerd is elke taal die door een NDPA wordt geaccepteerd noodzakelijk contextvrij. Voordat we deze eigenschap bewijzen geven we een voorbeeld van de constructie van een NDPA die eenvoudige rekenkundige expressies accepteert.

Zij $G = (N, T, P, E)$ een contextvrije grammatica met $N = \{E, T, F\}$, $T = \{a, b, c, (,), +, -, \times, /\}$ en met de volgende produktieregels in P

$$E \rightarrow T | E + T | E - T$$

$$T \rightarrow F | T \times F | T / F$$

$$F \rightarrow a | b | c | (E)$$

Uit hoofdstuk 2 weten we dat elke $x \in L(G)$ kan worden gegenereerd door een linkerafleiding. De NDPA die we construeren om $L(G)$ te accepteren simuleert deze linker afleidingen op zijn stack. Initieel is de inhoud van de stack $E \dashv$, hierbij is $\dashv$ een speciaal teken om de bodem van de stack aan te geven. Elk van de regels $E \rightarrow T, E \rightarrow E + T, E \rightarrow E - T$ kan nu worden toegepast om een nieuwe stackinhoud te genereren en deze zal respectievelijk bestaan uit $T \dashv, E + T \dashv$ of $E - T \dashv$. Bij elke stap die de automaat uitvoert wordt het topelement van de stack gehaald. Als het een nonterminaal symbool is dan kan het verder worden ontwikkeld door het toepassen van een produktieregel; het rechterlid wordt dan teruggeplaatst op de stack. Als het een terminaal symbool

is dan moet dit overeenkomen met het volgende inputsymbool, wil het proces kunnen worden voortgezet.
We definiëren nu $M = (K, T, V, p, k_1, \dashv, \{k_3\})$ met

$$K = \{k_1, k_2, k_3\},$$
$$V = N \cup T \cup \{\dashv\},$$

en p wordt gedefinieerd door

$$\begin{aligned}
p(k_1, \dashv, \varepsilon) &= \{(k_2, E \dashv)\},\\
p(k_2, E, \varepsilon) &= \{(k_2, T), (k_2, E + T), (k_2, E - T)\},\\
p(k_2, T, \varepsilon) &= \{(k_2, F), (k_2, T \times F), (k_2, T/F)\},\\
p(k_2, F, \varepsilon) &= \{(k_2, a), (k_2, b), (k_2, c), (k_2, (E))\},\\
p(k_2, a, a) &= \{(k_2, \varepsilon\}, p(k_2, b, b)\\
&= \{(k_2, \varepsilon)\}, p(k_2, c, c) = \{(k_2, \varepsilon\},\\
p(k_2, +, +) &= \{(k_2, \varepsilon)\}, p(k_2, -, -) = \{(k_2, \varepsilon)\},\\
p(k_2, \times, \times) &= \{(k_2, \varepsilon)\}, p(k_2, /, /) = \{(k_2, \varepsilon)\},\\
p(k_2, (, () &= \{(k_2, \varepsilon)\}, p(k_2,),)) = \{(k_2, \varepsilon)\},\\
p(k_2, \dashv, \varepsilon) &= \{(k_3, \varepsilon)\}.
\end{aligned}$$

Toestand k_1 wordt gebruikt om de stack op $E \dashv$ te initialiseren en toestand k_3 wordt slechts bereikt als de stack alleen het symbool $\dashv$ bevat. De simulatie van de linkerafleiding wordt dus uitgevoerd in toestand k_2. In tabel 6.1 is een rij configuraties van M weergegeven om te laten zien hoe $a \times (b + c)$ wordt geaccepteerd.

Het ontwerp van een efficiënte ontleder voor rekenkundige expressies, gebaseerd op deze automaat is geen gemakkelijke taak. Het probleem is dat drie van de gebruikte regels voor p een keuzemogelijkheid voor elke volgende stap openlaten. Als $x \in T(M)$, dan weten we wel dat er een of andere rij configuraties moet bestaan die ervoor zorgt dat M x accepteert, maar er bestaan ook veel toegelaten rijen configuraties die niet leiden tot acceptatie van x. In de hoofdstukken 7 en 8 zullen we zien dat de grammatica die de taal genereert bepaalde eigenschappen moet hebben om dit probleem te voorkomen. Is aan deze eigenschappen voldaan dan kan wel een efficiënte ontleder worden geconstrueerd.

Stelling 6.1

Als L een contextvrije taal is dan bestaat er een nondeterministische stapelautomaat, M met $L = T(M)$.

Bewijs (schets):
Stel de taal $L \in T^*$ wordt gegenereerd door de contextvrije grammatica $G = (N, T, P, S)$. We definiëren dan

	Rij configuraties	
Resterende input	Toestand	Stackinhoud
$a \times (b + c)$	k_1	$\dashv$
$a \times (b + c)$	k_2	$E \dashv$
$a \times (b + c)$	k_2	$T \dashv$
$a \times (b + c)$	k_2	$T \times F \dashv$
$a \times (b + c)$	k_2	$F \times F \dashv$
$a \times (b + c)$	k_2	$a \times F \dashv$
$\times (b + c)$	k_2	$\times F \dashv$
$(b + c)$	k_2	$F \dashv$
$(b + c)$	k_2	$(E) \dashv$
$b + c)$	k_2	$E) \dashv$
$b + c)$	k_2	$\neq + T) \dashv$
$b + c)$	k_2	$T + T) \dashv$
$b + c)$	k_2	$F + T) \dashv$
$b + c)$	k_2	$b + T) \dashv$
$+ c)$	k_2	$+ T) \dashv$
$c)$	k_2	$T) \dashv$
$c)$	k_2	$F) \dashv$
$c)$	k_2	$c) \dashv$
$)$	k_2	$) \dashv$
ε	k_2	$\dashv$
ε	k_3	ε

Tabel 6.1

$M = (K, T, V, p, k_1, \dashv, \{k_3\})$ zodanig dat
$K = \{k_1, k_2, k_3\}$,
$V = N \cup T \cup \{\dashv\}, \dashv \notin N \cup T$,

terwijl p wordt gedefinieerd als

$$p(k_1, \dashv, \varepsilon) = \{(k_2, S \dashv)\},$$
$$p(k_2, A, \varepsilon) = \{(k_2, \alpha) | A \to \alpha \text{ in } P\}, \text{ voor alle } A \in N,$$
$$p(k_2, a, a) = \{(k_2, \varepsilon)\}, \text{ voor alle } a \in T,$$

en

$$p(k_2, \dashv, \varepsilon) = \{(k_3, \varepsilon)\}.$$

Op grond van deze constructie geldt voor $x \in T^*, \alpha \in (N \cup T)^*$, dat $S \stackrel{*}{\Rightarrow} x\alpha$ een linkerafleiding is dan en slechts dan als $t((k_1, \dashv), x)$ $(k_2, \alpha \dashv)$ bevat. Dus geldt $x \in L(G)$ desd als $S \stackrel{*}{\Rightarrow} x$ desd als $t((k_1, \dashv), x)$ $(k_2, \dashv)$ bevat desd als $t((k_1, \dashv), x)$ $(k_3, \dashv)$ bevat desd als $x \in T(M)$.

Het omgekeerde van stelling 6.1 is ook waar, dat wil zeggen: iedere taal die door een NDPA wordt geaccepteerd is noodzakelijk contextvrij. Voordat we dit kunnen bewijzen moeten we een andere eigenschap van NDPA's afleiden. Stel $M = (K, T, V, p, k_1, A_1, F)$ is een willekeurige NDPA. We construeren nu uit M een NDPA, $M' = (K', T, V', p', k_1', \dashv, F')$ zodanig dat

$$K' = K \cup \{k_1', k_2'\} \qquad \text{met } k_1', k_2' \notin K,$$
$$V' = V \cup \{\dashv\} \qquad \text{met } \dashv \notin V,$$

p' is gelijk aan p, maar we voegen de volgende regels toe ten aanzien van de nieuwe toestanden k_1', k_2' en het nieuwe stacksymbool $\dashv$,

$$p'(k_1', \dashv, \varepsilon) = \{(k_1, A_1 \dashv)\},$$
$$p'(k, A, \varepsilon) = \{(k_2', \varepsilon)\}, \text{ voor alle } A \in V', k \in F,$$

en

$$p'(k_2', A, \varepsilon) = \{(k_2', \varepsilon)\}, \text{ voor alle } A \in V',$$
$$F' = \{k_2'\}.$$

Met de eerste van deze regels wordt het symbool dat de bodem van de stack aangeeft op de stack gezet voordat M' begint met de simulatie van M. Als M' ooit een eindtoestand in F bereikt, moet er minstens nog één symbool op de stack staan. De tweede regel kan worden gebruikt door M' om naar toestand k_2' over te gaan zonder dat hiervoor extra input nodig is. Omdat $k_2' \in F'$ geldt $T(M) = T(M')$. M' heeft maar één eindtoestand en zodra M' naar die toestand overgaat kan de derde regel worden gebruikt om de stack leeg te maken. We krijgen de volgende stelling.

Stelling 6.2

Als L wordt geaccepteerd door een NPDA, M, dan wordt L ook geaccepteerd door een NPDA, M', met maar één eindtoestand, waarin de automaat de stack leegmaakt zonder dat hiervoor verdere input nodig is. Deze eindtoestand is verder de enige toestand waarin de stack kan worden leeggemaakt.

Nu kunnen we de tweede belangrijke stelling van dit hoofdstuk formuleren.

Stelling 6.3

Als $L = T(M)$ voor een of andere nondeterministische stapelautomaat dan is L een contextvrije taal.

Bewijs (schets):
We mogen veronderstellen dat L wordt geaccepteerd door een NDPA met de eigenschappen die in stelling 6.2 werden beschreven. We labelen de toestanden $k_1, k_2, \ldots, k_n$ met k_1 als begintoestand en k_n als enige eindtoestand. Zij $M = (\{k_1, k_2, \ldots, k_n\}, T, V, p, k_1, A_1, \{k_n\})$ de NDPA die L accepteert. We moeten

nu uit M een contextvrije grammatica G construeren zodanig dat $T(M) = L(G)$.

Bij iedere $A \in V$ gebruiken we het nonterminale symbool A^{ij} in de geconstrueerde grammatica om alle inputstrings te genereren die ons van toestand k_i naar toestand k_j in M zouden brengen en tegelijkertijd verwijderen we het symbool A van de top van de stack. De rest van de stack laten we ongemoeid. $T(M)$ is nu de taal die door A_1^{1n} wordt gegenereerd.
We definiëren $G = (N, T, P, S)$ met

$$N = \{A^{ij} | A \in V, 1 \leq i, j \leq n\},$$
$$S = A_1^{1n}$$

P construeren we als volgt: bij iedere regel die de stapelfunctie van M definieert stellen we

(a) als $p(k_i, A, a), k_i \in K, A \in V, a \in T \cup \{\varepsilon\}$ $(k_j, B_1 B_2 \ldots B_m), k_j \in K, B_1, B_2, \ldots, B_m \in V$ bevat dan zit

$$A^{ij} \rightarrow aB_1^{in_1} B_2^{n_1 n_2} \ldots B_m^{n_{m-1} j}$$

in P voor alle $1 \leq n_1, n_2, \ldots, n_{m-1} \leq n$, en
(b) als $p(k_i, A, a), k_i \in K, A \in V, a \in T \cup \{\varepsilon\}$ $(k_j, \varepsilon), k_j \in K$ bevat dan zit $A^{ij} \rightarrow \alpha$ in P.

Uit deze constructie volgt dat $x \in T(M)$ desd als $t((k_1, A_1), x)$ (k_n, ε) bevat desd als $A^{1n} \overset{*}{\Rightarrow} x$ desd als $x \in L(G)$ en hiermee is de stelling bewezen.

Op deze wijze hebben we een andere omschrijving verkregen van contextvrije talen—het zijn de talen die door nondeterministische stapelautomaten worden geaccepteerd. Een aantal verdere eigenschappen van de klasse van contextvrije talen kunnen we op grond hiervan bewijzen.

Stelling 6.4

Als L een CFL is en R is een reguliere verzameling dan is ook $L \cap R$ een CFL.

Bewijs (schets):
$L = T(M)$ voor een NDPA, $M = (K, T, V, p, k_1, A_1, F)$, en $R = T(\tilde{M})$ voor een NFSA, $\tilde{M} = (\tilde{K}, T, \tilde{t}, \tilde{k_1}, \tilde{F})$. We construeren een NPDA $M \times \tilde{M}$ die $L \cap R$ accepteert. Het komt er op neer dat $M \times \tilde{M}$ het gedrag van M en $\tilde{M}$ tegelijkertijd simuleert en x wordt dan en slechts dan door $M \times \tilde{M}$ geaccepteerd als beide automaten afzonderlijk x accepteren.

Formeel geldt $M \times \tilde{M} = (K \times \tilde{K}, T, V, p', [k_1, \tilde{k_1}], A_1, F \times \tilde{F})$. p' wordt met behulp van de volgende regels gedefinieerd: voor $k, k' \in K, \tilde{k}, \tilde{k'} \in \tilde{K}, A \in V, a \in T, X \in V^*$ geldt

(a) $([k', \tilde{k'}], X) \in p'([k, \tilde{k}], A, a)$ desd als $(k', Z) \in p(k, A, a)$ en $\tilde{k'} \in \tilde{t}(\tilde{k}, a)$,
(b) $([k', \tilde{k'}], X) \in p'([k, \tilde{k}], A, \varepsilon)$ desd als $(k', Z) \in p(k, A, \varepsilon)$ en $\tilde{k'} = \tilde{k}$.

Met behulp van inductie naar het aantal stappen dat tijdens de berekening wordt uitgevoerd kan worden aangetoond dat $([k,\tilde{k}],X)$ in $t_{M\times\tilde{M}}(([k_1,\tilde{k_1}],A_1),x)$ zit desd als (k,X) in $t_M((k_1,A_1),x)$ zit en als $\tilde{k}$ in $\tilde{t}(\tilde{k_1},x)$ zit. Dus geldt $x \in T(M\times\tilde{M})$ desd als $t_{M\times\tilde{M}}(([k_1,\tilde{k_1}],A_1),x)$ $([k,\tilde{k}],X)$ bevat en voor een $k \in F, \tilde{k} \in \tilde{F}, X \in V^*$ en dit geldt desd als $f_M((k_1,A_1),x)$ (k,X) bevat voor $k \in F$ en $X \in V^*$ en als $\tilde{t}(\tilde{k_1},x)$ $\tilde{k},\tilde{k} \in \tilde{F}$ bevat desd als $x \in T(M)$ en $x \in T(\tilde{M})$. Dus geldt $T(M\times\tilde{M}) = T(M)\cap T(\tilde{M}) = L\cap R$ en hiermee is de stelling bewezen.

6.3 Deterministische stapelautomaten

Als een stapelautomaat hoogstens één stap kan maken vanuit iedere configuratie dan heet de automaat deterministisch. Formeler gesteld: een automaat $M = (K,T,V,p,k_1,A_1,F)$ is een *deterministische stapelautomaat* (Engels: Deterministic Pushdown Automaton (DPDA)) als voor alle $k \in K, a \in T$ en $A \in V$ geldt

(1) $p(k,A,a)$ bevat hoogstens één element,
(2) $p(k,A,\varepsilon)$ bevat hoogstens één element,
(3) als $p(k,A,\varepsilon) \neq \emptyset$ dan is $p(k,A,a) = \emptyset$ voor alle $a \in T$.

De derde eis zorgt ervoor dat er geen enkele andere stap kan worden gedaan vanuit dezelfde configuratie als er een stap kan worden gedaan zonder dat hiervoor input nodig is.

Op grond van de definitie van een DPDA hebben alle regels die de automaat definiëren rechterleden die ofwel singleton-verzamelingen zijn, ofwel leeg zijn. We spraken al eerder af dat we regels met lege rechterleden zouden weglaten en alle regels hebben dus de volgende vorm

$$f(k,A,a) = \{(k',X)\}.$$

De gebruikelijke notatie voor dit type regels is

$$f(k,A,a) = (k',X).$$

en deze notatie zullen wij hier ook verder aanhouden.

Een taal die door een DPDA wordt geaccepteerd heet een *deterministische contexvrije taal* (Engels: deterministic context-free language (DCFL)) of korter een *deterministische taal*. Helaas is niet iedere CFL deterministisch, maar de DCFL's vormen wel een belangrijke klasse van talen. Wordt een CFL geaccepteerd door een DPDA dan is de oplossing van het afleidingsprobleem een stuk eenvoudiger. De syntaxisdefinities van de meeste moderne programmeertalen zijn zo geformuleerd dat in die taal geschreven programma's betrekkelijk gemakkelijk kunnen worden gecompileerd. De efficiëntie van een ontleder (parser) hangt af van de exacte eigenschappen van de syntaxdefinitie van de taal. De allerzwakste randvoorwaarde is wel dat de syntaxis een deterministische taal

moet definiëren. In de volgende twee hoofdstukken zullen we zien waartoe deze en sterke randvoorwaarden kunnen leiden. De rest van dit hoofdstuk wijden we aan voorbeelden en eigenschappen van deterministische talen.

Eerst kijken we wat nader naar reguliere verzamelingen. Elke reguliere verzameling kan worden geaccepteerd door een deterministische eindige automaat en zo'n DFSA kan eenvoudig worden gesimuleerd door een DPDA die zijn stack niet gebruikt. Hieruit volgt dat iedere reguliere verzameling deterministisch is, maar niet iedere DCFL is regulier. Dit laatste tonen we aan met een voorbeeld van een DPDA die de niet reguliere taal $\{a^n b^n | n \geq 1\}$ accepteert.
$M = (\{k_1, k_2, k_3\}, \{a, b\}, \{a, \dashv\}, p, k_1, \dashv, \{k_3\})$ met

$$\begin{aligned} p(k_1, \dashv, a) &= (k_1, a \dashv), \\ p(k_1, a, a) &= (k_1, aa), \\ p(k_1, a, b) &= (k_2, \varepsilon), \\ p(k_2, a, b) &= (k_2, \varepsilon), \\ p(k_2, \dashv, \varepsilon) &= (k_3, \varepsilon). \end{aligned}$$

Zolang deze DPDA in toestand k_1 is wordt elke a op de stack gezet totdat een b wordt ingevoerd. Dan gaat de automaat over in toestand k_2 waarin elementen van de stack worden gehaald. Hierbij wordt voor elke ingevoerde b een a van de stack gehaald. Als nu het oorspronkelijk aantal a's op de stack precies overeenkomt met het aantal ingevoerde b's dan bevat de stack uiteindelijk alleen het bodemsymbool $\dashv$ en er is geen verdere input nodig om naar de eindtoestand over te gaan.

Het is zonder meer duidelijk dat iedere deterministische taal een contextvrije taal is; immers, de taal wordt geaccepteerd door een DPDA en dit is een nondeterministische stapelautomaat waaraan een bepaalde extra beperking is opgelegd. In oefening 5.12 lieten we zien dat er een CFL, $L \subset T^*$ bestaat waarvan het complement $\overline{L} = T^* \backslash L$ geen CFL is. Hier zullen we nu aantonen dat het complement van een DCFL noodzakelijk ook een DCFL is. Het bewijs dat we schetsen is vrij ingewikkeld en kan bij eerste lezing worden overgeslagen. Een belangrijke gevolgtrekking is dat er contextvrije talen bestaan die niet deterministisch zijn.

Zij $L \subset T^*$ een deterministische taal gegenereerd door de DPDA, $M = (K, T, V, p, k_1, A_1, F)$. Wat we willen is in principe het omkeren van de rollen van de eindtoestanden en de inwendige toestanden zodat een DPDA, $\overline{M}$, ontstaat die de taal $\overline{L} = T^* \backslash L$ accepteert. Voordat we zover zijn moeten we eerst een paar problemen oplossen.

Het eerste probleem is dat er in M 'doodlopende stukken' kunnen zitten, dat wil zeggen: er kunnen strings $x \in T^*$ bestaan zodanig dat het invoeren van een prefix van x in M leidt tot een configuratie van waaruit geen volgende stap mogelijk is, of van waaruit een oneindige rij ε-stappen kan worden uitgevoerd. In beide gevallen geldt toch $x \in \overline{L}$ en we moeten dus $\overline{M}$ met zorg construeren willen dit soort strings worden geaccepteerd. Dit probleem lijkt op het probleem dat we bij het bewijs van stelling 3.4 tegenkwamen toen we aantoonden dat

het complement van een reguliere verzameling noodzakelijk regulier is. Net als bij dat bewijs kunnen we het probleem oplossen door de oorspronkelijke automaat op een geschikte manier aan te passen. We construeren dus uit M een nieuwe DPDA, M', zó dat $T(M) = T(M')$, maar ook zo dat M' altijd de gehele input inleest. M' is identiek aan M, maar kent een symbool $\dashv \notin V$ dat de bodem van de stack aangeeft en drie extra toestanden $s, d, f \notin K$. M' begint in toestand s met $\dashv$ op de stack, maakt vervolgens onmiddellijk een ε-stap naar toestand k_1 en plaatst $A_1 \dashv$ op de stack. M' gaat nu verder met het simuleren van M, maar met enkele belangrijke verschillen. Als voor M geen volgende stap mogelijk is gaat M' over in de dummytoestand d en leest van daaruit de rest van de input in. De extra eindtoestand f wordt gebruikt om de situatie te doorbreken waarin M een oneindige rij ε-stappen kan uitvoeren. Als zo'n rij mogelijk is in een of andere configuratie c dan moet er een configuratie $c' = (k, AX)$ bestaan met $k \in K, A \in V \cup \{\dashv\}, X \in (V \cup \{\dashv\})^*$ die vanuit c te bereiken is via een eindige rij ε-stappen, terwijl vanuit c' een oneidige rij ε-stappen kan worden uitgevoerd zonder dat verder elementen van de stack die X bevat worden gehaald. Tijdens deze oneindige rij stappen kan al dan niet een eindtoestand worden bereikt. Als wel een eindtoestand wordt bereikt, dan definiëren we M' zo dat de configuratie $c' = (k, AX)$ overgaat in (f, AX) na een input ε en als geen eindtoestand wordt bereikt dan laten we M' overgaan naar (d, AX) na deze input. Om ervoor te zorgen dat M' de hele input inleest, moeten we niet vergeten naar toestand d over te gaan zodra we toestand f hebben bereikt.

Formeler definiëren we

$$M' = (K', T, V', p', s, \dashv, F')$$

met

$$K' = K \cup \{s, d, f\} \text{ zodanig dat } s, d, f \notin K,$$
$$V' = V \cup \{\dashv\} \text{ zodanig dat } \dashv \notin V,$$
$$F' = F \cup \{f\}.$$

p' wordt als volgt gedefinieerd

(a) $p'(s, \dashv, \varepsilon) = (k_1, A_1 \dashv)$,
(b) voor alle $k \in K, A \in V$ en $a \in T$ met $p(k, A, a) = \emptyset$ en $p(k, A, \varepsilon) = \emptyset$ geldt $p'(k, A, a) = (d, A)$,
(c) $p'(d, A, a) = (d, A)$ voor alle $A \in V'$ en $a \in T$,
(d) voor alle $k \in K, A \in V$ waarvoor een oneindige rij ε-stappen kan worden gedaan vanuit een begintoestand (k, A) stellen we

$$p'(k, A, \varepsilon) = \begin{cases} (d, A) & \text{als geen van die stappen } M \\ & \text{in een eindtoestand brengt} \\ (f, A) & \text{in alle andere gevallen.} \end{cases}$$

(e) $p'(f, A, \varepsilon) = (d, A)$ voor alle $A \in V'$,

(f) als $p'(k, A, a)$ niet is gedefinieerd door één van de bovenstaande regels (a) tot en met (d) dan stellen we $p'(k, A, a) = p(k, A, a)$ met $k \in K, A \in V$ en $a \in T \cup \{\varepsilon\}$.

Ga na dat nu $T(M) = T(M')$ en dat M' altijd de gehele input leest. Misschien vraagt u zich af hoe nu eigenlijk de machine kan worden geconstrueerd. Vooral stap (d) geeft moeilijkheden: kunnen we vanuit een configuratie (k, A) nagaan of er een oneindige rij ε-stappen kan worden uitgevoerd en, zo ja, kunnen we te weten komen of de DPDA tijdens die stappen in een eindtoestand terechtkomt? Het antwoord op beide vragen is 'ja'—zie oefening 6.11 aan het einde van dit hoofdstuk.

Voordat we de rollen van de eindtoestanden en de inwendige toestanden kunnen omkeren moeten we nog een laatste probleem oplossen. Dit probleem ontstaat als M' een aantal ε-stappen uitvoert na een input $x \in T^*$. Na bepaalde stappen kan M' zich in een eindtoestand bevinden, na bepaalde andere juist niet. Als we de rollen van eindtoestanden en niet-eindtoestanden omkeren dan verandert dit niets aan deze situatie. We moeten het probleem oplossen door opnieuw een DPDA, M'' uit M' te construeren met $T(M'') = T(M') = T(M)$. M'' heeft dezelfde toestanden als M' maar hierbij noteren we extra informatie die aangeeft of de automaat langs een eindtoestand is gekomen sinds de laatste niet-lege input. Als dit het geval is dan is M'' in een toestand met suffix 1 en als dit niet het geval is dan bevindt M'' zich in een toestand met suffix 2. Tijdens het lezen van een niet-lege input door M'' kan deze overgaan in een toestand met suffix 1 naar een toestand met suffix 2 of omgekeerd, afhankelijk van het antwoord op de vraag of M' een eindtoestand is gepasseerd.

Formeler: als $M' = (K', T, V', p', s, \dashv, F')$, dan definiëren we $M'' = (K'', T, V', p'', s'', \dashv, F'')$ met

$$K'' = \{k_i | k \in K', i = 1 \text{ of } 2\},$$
$$s'' = \begin{cases} s_1 & \text{als } \varepsilon \in T(M), \\ s_2 & \text{in alle andere gevallen,} \end{cases}$$
$$F'' = \{k_1 | k \in K'\}.$$

p'' definiëren we als volgt:

(a) als $p'(k, A, \varepsilon) = (k', X), k' \in K', X \in (V')^*$ voor $k \in K'$ en $A \in V'$ dan is

$$p''(k_1, A, \varepsilon) = (k_1'', X) \text{ en}$$
$$p''(k_2, A, \varepsilon) = \begin{cases} (k_1', X) & \text{als } k' \in F', \\ (k_2', X) & \text{in alle andere gevallen.} \end{cases}$$

(b) voor $k \in K', A \in V'$ en $a \in T$, als $p'(k, A, a) = (k', X), k' \in K', X \in (V')^*$ dan is

$$p''(k_1, A, a) = p''(k_2, A, a) = \begin{cases} (k_1', X) & \text{als } k' \in F', \\ (k_2', X) & \text{in andere gevallen.} \end{cases}$$

Gesloten onder	Regulier	Deterministisch	Contextvrij
vereniging	ja	nee	ja
concatenatie	ja	nee	ja
complement	ja	ja	nee
doorsnede	ja	nee	nee
Kleene afsluiting	ja	nee	ja
doorsnede met reguliere verzameling	ja	ja	ja
homomorfisme	ja	nee	ja

Tabel 6.2: Eigenschappen ten aanzien van geslotenheid

Nu kunnen we veilig de rollen van de eindtoestanden en de niet-eindtoestanden van M'' omkeren en we krijgen dan een DPDA $\overline{M}$ die $\overline{L}$ accepteert en hiermee hebben we de schets van het bewijs van de volgende stelling voltooid.

Stelling 6.5

Als $L \subset T^*$ een determistische taal is dan is $\overline{L} = T^* \backslash L$ dat ook.

Ook de volgende stelling hebben we hiermee bewezen.

Stelling 6.6

(a) Iedere reguliere taal is deterministisch, maar er bestaan deterministische talen die niet regulier zijn.
(b) Iedere deterministische taal is contextvrij, maar er bestaan contextvrije talen die niet deterministisch zijn.

De eigenschappen van deterministische talen verschillen aanmerkelijk van die van contextvrije talen. We gaven een schets van een bewijs dat deterministische talen gesloten zijn onder de complement-operatie. Dit wil zeggen dat het feit dat L deterministisch is impliceert dat $\overline{L}$ deterministisch is, maar dit is niet de enige eigenschap die verschillend is voor deze twee klassen van talen. In tabel 6.2 is een aantal eigenschappen ten aanzien van geslotenheid, voorzover bekend aangegeven voor reguliere verzamelingen deterministische talen en contextvrije talen.

6.4 Oefeningen

1. Gegeven een input 012345 en een stack S. Gevraagd: welke van de volgende strings kunnen louter met PUSH en POP operaties op S worden gegenereerd?
   (a) 543210,
   (b) 534210,
   (c) 431250,
   (d) 415320,
   (e) 542301.

   (Algemeen: als $x = a_1 a_2 \ldots a_n$, waarbij $a_1, \ldots, a_n$ verschillende symbolen zijn dan zijn er $n!$ verschillende permutaties mogelijk. Daarvan zijn er echter slechts $\frac{1}{n+1}\binom{2n}{n}$ bereikbaar door via een stack alleen PUSH en POP operaties toe te passen. Zie bijvoorbeeld *Graph Algorithms* door Shimon Even, Pitman Publishing Ltd. voor een eenvoudig en elegant bewijs.)
2. Construeer DPDA's om aan te tonen dat de volgende talen deterministisch zijn.
   (a) $\{wcw^r | w \in \{a,b\}^*\}$,
   (b) $\{a^i b^j | j > i\}$.
3. Als L deterministisch is en R regulier bewijs dan dat $L \cap R$ deterministisch is.
4. Een NPDA, $M = (K, T, V, p, k_1, A_1, F)$ accepteert een taal $E(M) \subset T^*$ met *lege opslag* als $E(M) = \{x | t_M((k_1, A_1), x)$ bevat (k, ε) voor een $k \in K\}$. Bewijs dat $E(M)$ noodzakelijk contextvrij is. (Het omgekeerde is ook waar: voor elke CFL, L, bestaat er een NPDA, M, met $L = E(M)$. Dit volgt uit stelling 6.2.)
5. Ontwerp een NPDA die strings accepteert die worden gegenereerd door een grammatica met de volgende produktieregels

$$S \rightarrow aA | aBB$$
$$A \rightarrow Ba | Sb$$
$$B \rightarrow bAS | \varepsilon$$

6. L is een deterministische taal. Bewijs dat $MIN(L) = \{x | x \in L$ en geen $w \in L$ is een zuivere prefix van $x\}$ ook deterministisch is.
7. Maak gebruik van het feit dat $\{a^i b^j c^k | i \neq j$ en $j \neq k\}$ geen CFL is om te bewijzen dat $\{a^i b^j c^k | i = j$ of $j = k\}$ niet deterministisch is. Toon vervolgens aan dat er twee deterministische talen L_1 en L_2 bestaan zodanig dat $L_1 \cup L_2$ niet deterministisch is.
8. Gebruik het resultaat van oefening 6.7 om te bewijzen dat deterministische talen niet gesloten onder doorsnede zijn.

9. Toon aan dat iedere DCFL wordt geaccepteerd door een DPDA, M, zodanig dat geen enkele stap meer dan twee symbolen uit het stackalfabet op de stack plaatst.
10. Bewijs dat iedere DCFL, $L \subset T^*$, wordt geaccepteerd door een DPDA, $M = (K, T, V, p, k_1, A_1, F)$, zodanig dat als er een stap wordt gedefinieerd door $p(k, A, a) = (k', X), k, k' \in K, a \in T \cup \{\varepsilon\}, A \in V, X \in V^*$ deze stap noodzakelijk van één van de volgende typen is
    (a) een verwijdering van de stack, d.w.z. $X = \varepsilon$,
    (b) een plaatsing op de stack van een enkel symbool, d.w.z. $X = BA$, $A, B \in V$, of
    (c) een stap waarbij de stack onveranderd blijft, d.w.z. $X = A$.
11. Beschouw een DPDA, $M = (\{k_1, \ldots, k_n\}, T, \{A_1, \ldots, A_m\}, p, k_1, A_1, F)$ die de eigenschappen heeft die in oefening 6.10 werden beschreven. Definieer drie Boole'se 3-dimensionale arrays B_1, B_2, B_3 met

    $B_1[i, j, k] = true$ desd als $t_M((k_i, A_k), \varepsilon)$ (k_j, A_k) bevat,
    $B_2[i, j, k] = true$ desd als $t_M((k_i, A_k), \varepsilon)$ (k_j, ε) bevat,
    $B_3[i, j, k] = true$ desd als $t_M((k_i, A_k), \varepsilon)$ (k_j, X) bevat voor een $X \in V^*$.

    Formuleer iteratieve algoritmen voor de constructie van deze arrays vanuit M. Toon vervolgens aan dat stap (d) in de constructie van M' in dit hoofdstuk effectief berekenbaar is.

Hoofdstuk 7

Neerwaartse ontleding

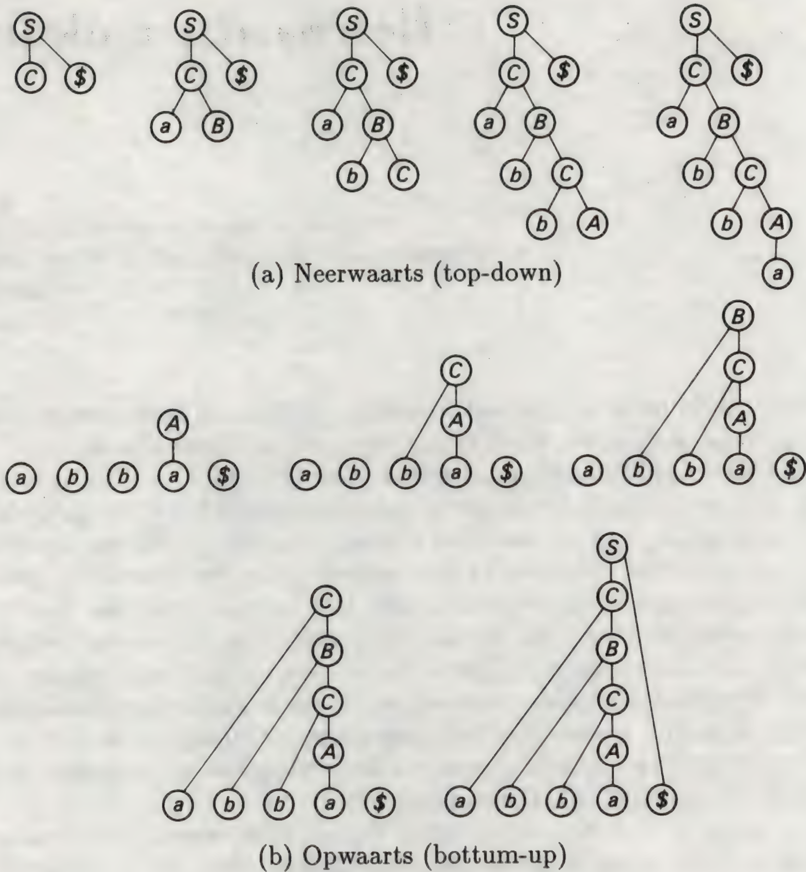
Iam nova progenies caelo demittitur alto
(Een nieuwe generatie daalt nu af van boven)
—Vergilius
Eclogae IV

In hoofdstuk 6 lieten we zien dat nondeterministische stapelautomaten contextvrije talen accepteren. Juist wegens het nondeterministische karakter van deze machines is tijdens het ontleden van zo'n contextvrije taal 'backtracking' nodig. Het ontleedalgoritme voert op deterministische wijze stappen uit en kiest een bepaalde stap als meer dan één stap mogelijk is. Dit kan betekenen dat bij een verkeerde keuze één of meer stappen terug moeten worden gedaan om vervolgens opnieuw een keuze te maken. Dit laatste noemt men 'backtracking' of 'terugkrabbelen'. Als we aan een grammatica die een taal definieert bepaalde beperkingen opleggen dan is het wel mogelijk efficiënte op een stack gebaseerde ontleedalgoritmen te ontwerpen die van bepaalde standaardtechnieken gebruik maken. Ontwerpers van programmeertalen zijn op de hoogte van deze beperkingen en trachten de grammatica van hun taal zo te formuleren dat een snelle en efficiënte compiler mogelijk wordt.

Een string $x = a_1a_2\ldots a_n \in L(G)$ kan op twee verschillende manieren worden ontleed. Ten eerste kunnen we een afleidingsboom van boven af of *top-down* construeren; we beginnen dan bij de wortel S en bouwen de boom zo van bovenaf op dat de blaadjes de symbolen $a_1, a_2, \ldots, a_n$ zijn. Ten tweede kunnen we van onderen, of *bottom-up* beginnen; we gaan dan uit van de string x en proberen de inwendige knopen van de boom tot aan de wortel af te leiden. In figuur 7.1 zijn deze twee benaderingen schematisch weergegeven voor de string *abba\$* en de contextvrije grammatica G_1 met regels

$$S \rightarrow C\$$$
$$C \rightarrow bA|aB$$
$$A \rightarrow a|aC$$
$$B \rightarrow b|bC$$

In dit hoofdstuk richten wij onze aandacht op 'LL-ontleding', een top-down methode. In hoofdstuk 8 bespreken wij opwaartse of bottom-up ontleedmethoden. Om op een taal een LL-ontleedmethode te kunnen toepassen, moeten er



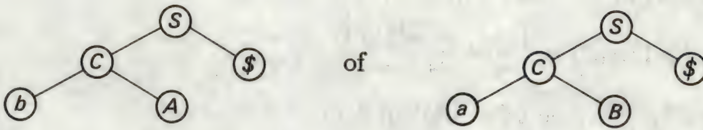
Figuur 7.1

vrij strenge restricties aan de grammatica die de taal definieert worden opgelegd. In praktische situaties blijken deze restricties, die ons beperken tot een subklasse van de deterministische talen, niet erg belemmerend want vrijwel alle moderne compilers berusten op de bijbehorende ontleedmethode.

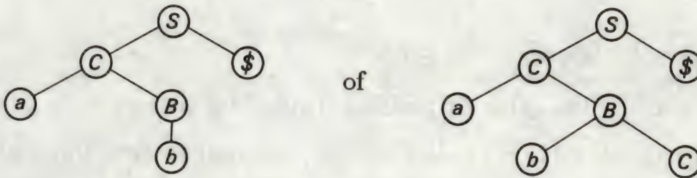
7.1 LL(K)-grammatica's

We bekijken weer de contextvrije grammatica G_1. De neerwaartse constructie van de ontleedboom voor de string *abba$* is in figuur 7.1a getekend. Hoe wordt zo'n boom nu opgebouwd? We beginnen met knoop Ⓢ en we kunnen maar één produktieregel toepassen. We krijgen dan de volgende boom:  . We

kunnen nu Ⓒ uitbreiden en krijgen



afhankelijk van het kiezen van de regel $C \rightarrow bA$ of $C \rightarrow aB$. Omdat *abba\$* met een *a* begint moet de tweede mogelijkheid de juiste zijn; met andere woorden: door naar het eerste symbool van de input te kijken kunnen we bepalen welke regel we moeten toepassen. We hebben nu een *a* gegenereerd die overeenkomt met de eerste *a* van de inputstring en nu moeten we controleren of $B \stackrel{*}{\Rightarrow} bba$. De knoop B kunnen we weer op twee manieren uitbreiden. We krijgen:



Nu kunnen we niet op grond van dit resultaat beslissen welke van deze mogelijkheden de juiste is, maar als we twee stappen vooruit kijken dan gaat het wel. Als de volgende twee symbolen die we met de inputstring moeten vergelijken *b\$* zijn dan moeten we het eerste alternatief kiezen, zijn ze *bb* of *ba* dan moeten we het tweede alternatief kiezen. In ons geval weten we dat de volgende twee symbolen *bb* zijn en we kiezen dus voor de tweede mogelijkheid. Nu moeten we C verder uitbreiden en controleren dat $C \stackrel{*}{\Rightarrow} ba$. We hoeven slechts een symbool vooruit te kijken om te beslissen dat we de regel $C \rightarrow ba$ moeten gebruiken en vervolgens moeten we nog eens twee symbolen vooruit kijken om de laatste produktieregel $A \rightarrow a$ te kiezen. De constructie van de afleidingsboom komt dus overeen met de linker afleiding $S \Rightarrow C\$ \Rightarrow aB\$ \Rightarrow abC\$ \Rightarrow abbA\$ \Rightarrow abba\$$.

G_1 is een voorbeeld van een (strenge) $LL(2)$-grammatica: men moet maximaal twee symbolen vooruitkijken om uit de mogelijke produktieregels de juiste te kiezen in elk stadium van de linkerafleiding. Een *$LL(k)$-grammatica* ($k \geq 1$) is een grammatica waarbinnen, gegeven een zin $wA\gamma, w \in T^*, A \in N, \gamma \in (N \cup T)^*$ gegenereerd door een linkerafleiding, hoogstens k symbolen vooruit moet worden gekeken om eenduidig vast te stellen welke van de produktieregels met A in het linkerlid moet worden gebruikt. De eerste L in LL staat voor 'lezen van links naar rechts' en de tweede L staat voor 'linkerafleiding'. Binnen een $LL(k)$-grammatica kan de keuze van de toe te passen regel dus niet alleen afhangen van het nonterminale symbool A en van de volgende k te bekijken inputsymbolen, maar ook van de string $w \in T^*$ voor A in de zin en van de string $\gamma \in (N \cup T)^*$ na A in de zin. Hangt de volgende produktieregel alleen af van het uit te breiden nonterminale symbool en van de volgende k te bekijken symbolen dan noemen we de grammatica *streng* $LL(k)$.

Om deze begrippen formeel te definiëren, voeren we in:

$$\text{EERSTE}_k : T^* \to T^* \text{ met}$$

$$\text{EERSTE}_k(x) = \begin{cases} x & \text{als } |x| \leq k, \\ y & \text{als } x = yz, y \in T^k, z \in T^*. \end{cases}$$

We laten EERSTE_k op talen werken door te definiëren

$$\text{EERSTE}_k(L) = \{\text{EERSTE}_k(x) | x \in L\} \text{ voor alle } L \subset T^*.$$

Nu kunnen we $LL(k)$ en streng $LL(k)$ formeel definiëren.

Als $G = (N, T, P, S)$ een contextvrije grammatica is dan heet G $LL(k)$, $(k \geq 1)$ als geldt dat, wanneer er twee linkerafleidingen bestaan

$$S \stackrel{*}{\Rightarrow} wA\gamma \Rightarrow w\alpha\gamma \stackrel{*}{\Rightarrow} wy \in T^*$$

en

$$S \stackrel{*}{\Rightarrow} wA\gamma \Rightarrow w\beta\gamma \stackrel{*}{\Rightarrow} wz \in T^*$$

dat dan uit $\text{EERSTE}_k(y) = \text{EERSTE}_k(z)$ volgt dat $\alpha = \beta$.

G heet *streng* $LL(k)(k \geq 1)$ als geldt dat, wanneer er twee linkerafleidingen bestaan

$$S \stackrel{*}{\Rightarrow} wA\gamma \Rightarrow w\alpha\gamma \stackrel{*}{\Rightarrow} wy \in T^*$$

en

$$S \stackrel{*}{\Rightarrow} xA\delta \Rightarrow x\beta\delta \stackrel{*}{\Rightarrow} xz \in T^*$$

dat dan uit $\text{EERSTE}_k(y) = \text{EERSTE}_k(z)$ volgt dat $\alpha = \beta$.

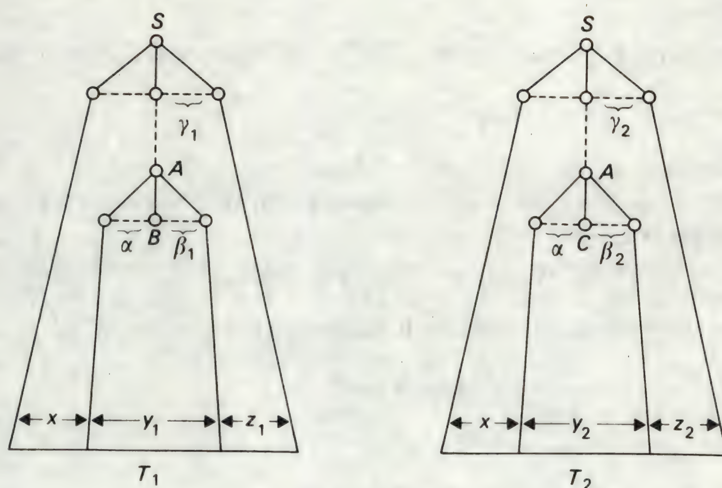
In het algemeen is het onmogelijk vast te stellen of een contextvrije grammatica meerduidig (ambigu) is of niet. Als we kunnen aantonen dat een grammatica streng $LL(k)$ is, dan bewijzen we in de volgende stelling dat hij eenduidig moet zijn. Dit geldt trouwens ook voor gewone $LL(k)$-grammatica's, (zie oefening 7.7).

Stelling 7.1

Als $G = (N, T, P, S)$ een strenge $LL(k)$-grammatica is, $(k \geq 1)$ dan is G eenduidig.

Bewijs:
Stel G is meerduidig. Dan bestaat er een $w \in L(G)$ met twee verschillende linkerafleidingen vanuit S. We zullen laten zien dat dit tot een tegenspraak leidt.

Stel T_1 en T_2 zijn de afleidingsbomen bij de twee verschillende linkerafleidingen van $w = a_1a_2 \ldots a_n$. Beide bomen hebben wortel S en blaadjes $a_1, a_2, \ldots, a_n$, maar zij zijn per definitie verschillend. We lopen nu door beide bomen door middel van een zogenaamde preorder bewandeling die als volgt recursief wordt gedefinieerd:



Figuur 7.2

bezoek de wortel;
bezoek de subbomen verbonden met de wortel van links naar rechts volgens de preorder methode.

Bij deze systematische bewandeling van de bomen moeten we knopen in T_1 en T_2 tegenkomen die verschillende labels hebben. Stel de knopen B en C zijn de eerste verschillende knopen die we tegenkomen, (zie figuur 7.2). Het zijn niet de wortels, want die hebben beide het label S. Zij hebben dus beide een ouder, laten we die A noemen.

De afleidingsboom T_1 komt nu overeen met de linkerafleiding

$$S \stackrel{*}{\Rightarrow} xA\gamma_1 \Rightarrow x\alpha B\beta_1\gamma_1 \stackrel{*}{\Rightarrow} xy_1\gamma_1 \stackrel{*}{\Rightarrow} xy_1z_1 = w$$

en T_2 komt overeen met

$$S \stackrel{*}{\Rightarrow} xA\gamma_2 \Rightarrow x\alpha C\beta_2\gamma_2 \stackrel{*}{\Rightarrow} xy_2\gamma_2 \stackrel{*}{\Rightarrow} xy_2z_2 = w$$

Omdat $\text{EERSTE}_k(y_1z_1) = \text{EERSTE}_k(y_2z_2)$ en $\alpha B\beta_1 \neq \alpha C\beta_2$ volgt dat G niet streng $LL(k)$ kan zijn en dit is de gezochte tegenspraak. Hiermee is de stelling dus bewezen.

Per definitie is iedere strenge $LL(k)$-grammatica ook $LL(k)$, maar voor $k > 1$ bestaan er $LL(k)$-grammatica's die niet streng zijn, (zie oefening 7.5). Voor $k = 1$ zijn de twee definities wel equivalent, (zie hierna). Ook is aangetoond, (D.J. Rosenkrantz & R.E. Stearns, [1970] Properties of deterministic top-down grammars, *Information and Control*, **17**, 226-256) dat de klasse van talen gegenereerd door $LL(k)$-grammatica's juist de klasse van talen gegenereerd door strenge $LL(k)$-grammatica's is, voor alle $k \geq 1$.

Stelling 7.2

De contextvrije grammatica $G = (N, T, P, S)$ is streng $LL(1)$ dan en slechts dan als zij $LL(1)$ is.

Bewijs:

$\Rightarrow$: Dit volgt onmiddellijk uit de definitie.

$\Leftarrow$: Zij G een niet strenge $LL(1)$-grammatica. We proberen weer een tegenspraak te vinden.

Omdat G niet streng $LL(1)$ is bestaan er per definitie verschillende produktieregels $A \rightarrow \alpha$ en $A \rightarrow \beta$ zó dat voor een $w, x, y_1, y_2, z_1, z_2 \in T^*$ en $\gamma, \delta \in (N \cup T)^*$ twee verschillende linkerafleidingen

$$S \overset{*}{\Rightarrow} wA\gamma \Rightarrow w\alpha\gamma \overset{*}{\Rightarrow} wy_1\gamma \overset{*}{\Rightarrow} wy_1y_2$$

en

$$S \overset{*}{\Rightarrow} xA\delta \Rightarrow x\beta\delta \overset{*}{\Rightarrow} xz_1\delta \overset{*}{\Rightarrow} xz_1z_2$$

bestaan met $\text{EERSTE}_1(y_1y_2) = \text{EERSTE}_1(z_1z_2)$.
We moeten aantonen dat G niet $LL(1)$ is. Er geldt ofwel

(a) $y_1 = z_1 = \varepsilon$ en dan is G zeker niet $LL(1)$, ofwel
(b) $y_1z_1 \neq \varepsilon$.

In geval (b) mogen we zonder verlies van algemeenheid veronderstellen dat $y_1 \neq \varepsilon$. Dan is dus $\text{EERSTE}_1(y_1y_2) = \text{EERSTE}_1(y_1) = \text{EERSTE}_1(z_1z_2)$. Maar de twee linkerafleidingen

$$S \overset{*}{\Rightarrow} xA\delta \Rightarrow x\alpha\delta \overset{*}{\Rightarrow} xy_1\delta \Rightarrow xy_1z_2$$

en

$$S \overset{*}{\Rightarrow} xA\delta \Rightarrow x\beta\delta \overset{*}{\Rightarrow} xz_1\delta \Rightarrow xz_1z_2$$

voldoen aan $\text{EERSTE}_1(y_1) = \text{EERSTE}_1(z_1z_2)$, maar $\alpha \neq \beta$ en dus is G niet $LL(1)$. Dit is de gezochte tegenspraak en hiermee is de stelling bewezen.

De speciale klasse van $LL(k)$-grammatica's met $k = 1$ is van bijzonder belang voor de compilerbouw. Ontwerpers van programmeertalen streven er vaak naar de syntaxis zo te definiëren dat deze grotendeels $LL(1)$ is. Zelfs als de grammatica niet is ontworpen met dit uitgangspunt, dan nog kunnen de regels vaak zo worden getransformeerd dat een gelijkwaardige $LL(1)$-vorm wordt bereikt. Een manier om dit voor elkaar te krijgen, is door gebruik te maken van het syntaxisverbeterende hulpmiddel van Foster, (Engels: Fosters's Syntax Improving Device (SID)). Dit werd al in 1968 beschreven in het *Computer Journal.* SID converteert automatisch iedere ingevoerde grammatica naar een $LL(1)$-vorm, voor zover dit mogelijk is. Vaak zijn er bepaalde gedeelten in de grammatica die niet in $LL(1)$-vorm kunnen worden gedefinieerd en die delen moeten dan afzonderlijk worden behandeld. Als voor het grootste deel van de syntaxis een

$LL(1)$-grammatica is opgesteld dan kan een ontleder gemakkelijk worden ontworpen of zelfs automatisch worden gegenereerd met behulp van een methode die recursieve afdaling wordt genoemd. De resterende grammaticaregels worden dan in de ontleder handmatig opgenomen. Deze techniek is bij bijna alle Pascal-compilers toegepast. Recursieve afdaling is niet alleen toepasbaar op $LL(1)$-grammatica's—het is een belangrijke techniek die leidt tot elegante en efficiënte ontleders voor tal van (deterministische) grammatica's. In de volgende paragraaf gaan we hier nader op in.

Eerst ontwikkelen we nu een stelling die leidt tot een algoritme waarmee je kunt testen of een willekeurige contextvrije grammatica G al dan niet (streng) $LL(1)$ is. We doen dit door uit G een grammatica G' te construeren met $L(G') = \{x\$|x \in L(G)\}$. Hierbij geeft het $\$$-teken het einde van de input aan. Als $G = (N, T, P, S)$ een willekeurige CFG is dan definiëren we de *vermeerderde grammatica* $G' = (N', T', P', S')$ met

$$N' = N \cup \{S'\} \text{ is zodanig dat } S' \notin N,$$
$$T' = T \cup \{\$\} \text{ is zodanig dat } \$ \notin T$$

en

$$P' = P \cup \{S' \to S\$\}.$$

Duidelijk is dat $L(G') = \{x\$|x \in L(G)\}$.

We definiëren nu de volgende functies op $N' \cup T'$.

$$\text{LEEG}(X) = \begin{cases} \textit{waar} & \text{als } X \overset{*}{\Rightarrow} \varepsilon, \\ \textit{onwaar} & \text{in andere gevallen.} \end{cases}$$

$$\text{EERSTE}(X) = \{a|a \in T' \text{ en } X \overset{*}{\Rightarrow} ax \text{ voor een } x \in (T')^*\}$$

$$\text{VOLGENDE}(X) = \{a|a \in T' \text{ en } S' \overset{*}{\Rightarrow} xXay \text{ voor een } x \in T^* \text{ en } y \in (T')^*\}$$

Voor iedere $X \in N' \cup T'$ definiëren we dus twee deelverzamelingen van T': EERSTE(X) en VOLGENDE(X).

De functie LEEG wordt uitgebreid tot een functie op $(N' \cup T')^*$ door

$$\text{LEEG}(\varepsilon) = \textit{waar} \text{ en}$$

$$\text{LEEG}(X\alpha) = \text{LEEG}(X) \textit{ en } \text{LEEG}(\alpha), X \in N' \cup T', \alpha \in (N' \cup T')^*.$$

Als dus $\gamma \in (N' \cup T')^*$ dan is LEEG(γ) waar desd als $\gamma \underset{G'}{\overset{*}{\Rightarrow}} \varepsilon$ desd als $\gamma \underset{G}{\overset{*}{\Rightarrow}} \varepsilon$.

Tenslotte definiëren we de functie KIJKVOORUIT op de produktieregels van G als

$$\begin{aligned} &\text{KIJKVOORUIT}(A \to X_1X_2 \ldots X_n) \\ &\quad = \bigcup \{\text{EERSTE}(X_i)|1 \le i \le n \textit{ en } \text{LEEG}(X_1X_2 \ldots X_{i-1})\} \\ &\quad\quad \cup \textit{ als } \text{LEEG}(X_1X_2 \ldots X_n) \textit{ dan } \text{VOLGENDE}(A) \textit{ anders } \emptyset. \end{aligned}$$

Nu kunnen we de volgende stelling formuleren.

Stelling 7.3

De contextvrije grammatica $G = (N, T, P, S)$ is (streng) $LL(1)$ dan en slechts dan als voor ieder paar verschillende produktieregels $A \rightarrow \alpha$ en $B \rightarrow \beta$ in de grammatica met hetzelfde linkerlid geldt dat KIJKVOORUIT$(A \rightarrow \alpha) \cap$ KIJKVOORUIT$(A \rightarrow \beta) = \emptyset$.

We geven nu een voorbeeld hoe je met behulp van deze stelling kunt vaststellen of een CFG $LL(1)$ is of niet. Zij gegeven de grammatica $G_2 = (N, T, P, S)$ met $N = \{S, A, B\}, T = \{a, b, c, d\}$; P bestaat uit

$$S \rightarrow BA|AAd$$
$$A \rightarrow a|\varepsilon$$
$$B \rightarrow bA|cB$$

Voor de bijbehorende vermeerderde grammatica geldt $N' = N \cup \{S'\} = \{S, A, B, S'\}$, $T' = T \cup \{\$\} = \{a, b, c, d, \$\}$ en p' is

$$S' \rightarrow S\$$$
$$S \rightarrow BA|AAd$$
$$A \rightarrow a|\varepsilon$$
$$B \rightarrow bA|cB$$

LEEG(A) = *waar*, maar LEEG(X) = *onwaar* voor alle andere $X \in N' \cup T'$. Duidelijk is dat EERSTE$(a_i) = \{a_i\}$ voor alle $a_i \in T'$ en VOLGENDE(S') = VOLGENDE$(\$) = \emptyset$, want dit geldt altijd in een vermeerderde grammatica. Ook verschijnt S' in zo'n grammatica niet in het rechterlid van de produktieregels en dus komt EERSTE(S') niet voor in de berekening van de verzameling KIJKVOORUIT. We beperken daarom onze aandacht tot de berekening van EERSTE$(A_i), A_i \in N$ en VOLGENDE$(X), X \in N \cup T$.

De waarden EERSTE$(A_i), A_i \in N$ kunnen worden uitgedrukt als de oplossing van een stelsel vergelijkingen geconstrueeerd volgens de volgende regel.

$$\text{EERSTE}(A_i) = \bigcup \{\text{EERSTE}(X_{ij}) \mid \text{ er is een produktieregel } A \rightarrow X_{i1}X_{i2}\ldots X_{in} \text{ en ofwel } j = 1 \text{ ofwel } \text{LEEG}(X_{i1}X_{i2}\ldots X_{ij-1})\}.$$

Als we deze regel op ons voorbeeld toepassen kunnen we afleiden dat

$$\begin{aligned}\text{EERSTE}(S) &= \text{EERSTE}(B) \cup \text{EERSTE}(A) \cup \text{EERSTE}(A) \cup \{d\}\\ &= \text{EERSTE}(B) \cup \text{EERSTE}(A) \cup \{d\}\end{aligned}$$

$$\text{EERSTE}(A) = \text{EERSTE}(a) = \{a\}$$

en

$$\text{EERSTE}(B) = \text{EERSTE}(b) \cup \text{EERSTE}(c) = \{b, c\}.$$

Elementen in $N' \cup T'$	VERZAMELING EERSTE	VERZAMELING VOLGENDE
$\$$	$\{\$\}$	$\emptyset$
a	$\{a\}$	$\{a, d, \$\}$
b	$\{b\}$	$\{a, \$\}$
c	$\{c\}$	$\{b, c\}$
d	$\{d\}$	$\{\$\}$
S'	$\{a, b, c, d\}$[1]	$\emptyset$
S	$\{a, b, c, d\}$	$\{\$\}$
A	$\{a\}$	$\{a, d, \$\}$
B	$\{b, c\}$	$\{a, \$\}$

[1] Deze verzameling is volledigheidshalve opgenomen.
EERSTE(S') = EERSTE(S) $\cup$ *als* LEEG(S) *dan* $\{\$\}$ *anders* $\emptyset$.

Tabel 7.1

In het algemeen kan een dergelijk stelsel vergelijkingen worden opgelost met behulp van de iteratieve methode die we aan het eind van hoofdstuk 5 beschreven. In dit geval ligt de oplossing voor de hand. De verzamelingen EERSTE zijn in tabel 7.1 opgesomd.

De verzamelingen VOLGENDE kunnen op overeenkomstige wijze worden uitgedrukt als de oplossing van een vergelijkingenstelsel. Als $X \in N \cup T$, dan geldt

$$\begin{aligned}\text{VOLGENDE}(X) = & \bigcup \{\text{VOLGENDE}(A) \mid \text{er is een produktieregel van de} \\ & \qquad\qquad \text{vorm } A \to \alpha X \beta \textit{ en } \text{LEEG}(\beta)\} \cup \\ & \bigcup \{\text{EERSTE}(X_{ij}) \mid \text{er is een produktieregel} \\ & \qquad\qquad A_i \to \alpha X X_{i1} X_{i2} \ldots X_{in} \textit{ en } j = 1 \\ & \qquad\qquad \textit{of } \text{LEEG}(X_{i1} X_{i2} \ldots X_{ij-1})\}.\end{aligned}$$

Hieruit kunnen we met betrekking tot ons voorbeeld afleiden dat

$$\begin{aligned}\text{VOLGENDE}(a) &= \text{VOLGENDE}(A) \\ \text{VOLGENDE}(b) &= \text{VOLGENDE}(B) \cup \text{EERSTE}(A) \\ &= \text{VOLGENDE}(B) \cup \{a\} \\ \text{VOLGENDE}(c) &= \text{EERSTE}(B) = \{b, c\} \\ \text{VOLGENDE}(d) &= \text{VOLGENDE}(S) \\ \text{VOLGENDE}(S) &= \text{EERSTE}(\$) = \{\$\} \\ \text{VOLGENDE}(A) &= \text{VOLGENDE}(S) \cup \text{VOLGENDE}(B) \\ &\quad \cup \text{EERSTE}(A) \cup \text{EERSTE}(d) \\ &= \text{VOLGENDE}(S) \cup \text{VOLGENDE}(B) \cup \{a, d\} \\ \text{VOLGENDE}(B) &= \text{VOLGENDE}(S) \cup \text{VOLGENDE}(B) \cup \text{EERSTE}(A) \\ &= \text{VOLGENDE}(S) \cup \text{VOLGENDE}(B) \cup \{a\}\end{aligned}$$

produktieregels	Verzameling KIJKVOORUIT
$S \rightarrow BA$	$\{b, c\}$
$S \rightarrow AAd$	$\{a, d\}$
$A \rightarrow a$	$\{a\}$
$A \rightarrow \varepsilon$	$\{a, d, \$\}$
$B \rightarrow bA$	$\{b\}$
$B \rightarrow cB$	$\{c\}$

Tabel 7.2

Ook dit stelsel vergelijkingen kan worden opgelost met de in hoofdstuk 5 beschreven techniek en de oplossingen zijn weer in tabel 7.1 afgedrukt.

Nu we de verzamelingen EERSTE en VOLGENDE hebben berekend voor alle elementen in $N \cup T$, kunnen we vervolgens de verzamelingen KIJKVOORUIT bepalen voor iedere produktieregel in de grammatica. In tabel 7.2 zijn deze verzamelingen afgedrukt. Omdat KIJKVOORUIT($A \rightarrow a$) $\cap$ KIJKVOORUIT ($A \rightarrow \varepsilon$) $\neq \emptyset$ is de grammatica niet $LL(1)$.

Met behulp van bovenstaand algoritme kan men dus testen of een willekeurige contextvrije grammatica al dan niet $LL(1)$ is. Jammer genoeg is het niet mogelijk een algoritme te formuleren dat bepaalt of er voor een gegeven willekeurige CFG al dan niet een equivalente $LL(1)$ grammatica bestaat—dit is een voorbeeld van een onoplosbaar probleem. Dus als het algoritme SID niet in staat blijkt een willekeurige CFG in een equivalente $LL(1)$-vorm te transformeren dan wil dit nog niet zeggen dat zo'n grammatica niet bestaat.

7.2 Recursieve afdaling

Als G een $LL(k)$-grammatica is dan kun je voor $L(G)$ een ontleder (parser) schrijven die werkt volgens de methode van de recursieve afdaling. De methode gebruikt weliswaar niet expliciet een stack, maar wel impliciet omdat gebruik wordt gemaakt van recursieve procedures en daarvoor heb je stacks nodig. Op het eerste gezicht lijkt de hier beschreven methode dus wel te verschillen van die die in hoofdstuk 6 werd beschreven, maar in feite is het een soortgelijke methode.

Recursieve afdaling in zijn eenvoudigste vorm biedt ons de mogelijkheid een taalherkenner te schrijven voor de taal die door een $LL(1)$-grammatica $G = (N, T, P, S)$ wordt gegenereerd. In zo'n herkenner zit per symbool van $N \cup T$ een procedure die elke string die uit dat symbool valt af te leiden kan herkennen. Als $X \in N \cup T$ dan noemen we die procedure pX. Als $a \in T$ dan leest pa eenvoudig het volgende nog niet vergeleken invoersymbool en gaat na of dit een a is. Als $A \in N$ en de produktieregels in P met A in het linkerlid

produktieregels	Verzameling KIJKVOORUIT
$S \rightarrow aAB$	$\{a\}$
$S \rightarrow bS$	$\{b\}$
$A \rightarrow aAb$	$\{a\}$
$A \rightarrow bBc$	$\{b\}$
$B \rightarrow AB$	$\{a, b\}$
$B \rightarrow c$	$\{c\}$

Tabel 7.3

zijn $A \rightarrow \alpha_1, A \rightarrow \alpha_2 \ldots A \rightarrow \alpha_n$, dan kunnen we beslissen wat de regel is die in de volgende stap van de afleiding moet worden toegepast door het volgende te vergelijken symbool te lezen en dit te vergelijken met de KIJKVOORUIT verzamelingen bij de produktieregels. In de procedure pA hangt het dus af van het volgende te vergelijken symbool welke regel $A \rightarrow \alpha_i$ we als volgende regel gebruiken. Als $\alpha_i = X_{i1}X_{i2} \ldots X_{im}, X \in N \cup T, 1 \leq j \leq m$ dan roepen we aan $pX_{i1}; pX_{i2}; \ldots; pX_{im}$.

Als voorbeeld geven we de volgende grammatica G_3 die zonder twijfel $LL(1)$ is.

$$S \rightarrow aAB|bS$$
$$A \rightarrow aA|bB$$
$$B \rightarrow AB|c$$

In tabel 7.3 zijn de bijbehorende KIJKVOORUIT verzamelingen opgesomd.

We veronderstellen dat *volgendsymbool* een functie is die het mogelijk maakt te 'spieken' wat het volgende nog niet vergeleken symbool is zonder dat dit hierdoor al uit de invoerrij wordt verwijderd. Als er geen volgend symbool meer bestaat dan roept *volgendsymbool* een foutprocedure 'mis' aan. Een eenvoudige herkenner voor $L(G_3)$ die werkt met recursieve afdaling ziet er nu in een Pascal-achtige taal als volgt uit.

begin pS;*if* geen input meer *then* halt *else* mis *end*.

Hierbij is de procedure pS als volgt gedefinieerd.

```
procedure pS ≡ case volgendsymbool of
               'a': pa; pA; pB;
               'b': pb; pS;
               'c': mis
               end;
```

```
procedure pA≡ case volgendsymbool of
              'a': pa; pA; pb;
              'b': pb; pB; pc;
              'c': mis
              end;
procedure pB≡ case volgendsymbool of
              'a','b': pA; pB;
              'c': pc
              end;
procedure pa ≡ begin
              read(symbool); if symbool ≠ 'a' then mis
              end;
procedure pb ≡ begin
              read(symbool); if symbool ≠ 'b' then mis
              end;
procedure pc ≡ begin
              read(symbool); if symbool ≠ 'c' then mis
              end;
```

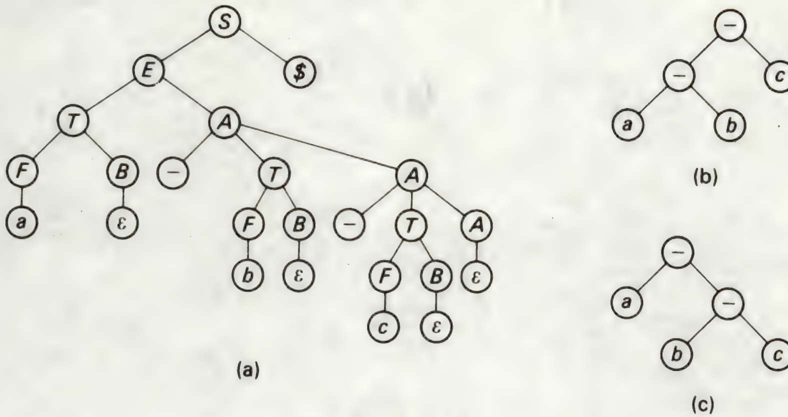
De procedure 'mis' stopt het proces en meldt dit.

De volgende stap is het invoegen van '*semantische handelingen*' in de procedures van de herkenner, dat wil zeggen het invoegen van bewerkingen die er voor zorgen dat een of andere toegelaten weergave van een geaccepteerde input als resultaat wordt opgeleverd. Als deze weergave bestaat uit een afleidingsboom dan hebben we meteen een algoritme waarmee het afleidingsprobleem kan worden opgelost. In de praktijk gebruiken compilerbouwers een geschiktere voorstellingswijze, bijvoorbeeld de semantische bomen die we in hoofdstuk 2 beschreven. We illustreren een en ander met een grammatica die rekenkundige uitdrukkingen genereert over de variabelen a, b en c. Elke uitdrukking wordt beeindigd door het 'einde-invoerteken' $\$$.

$$S \to E\$$$
$$E \to T|E+T|E-T$$
$$T \to F|T \times F|T/F$$
$$F \to a|b|c|(E)$$

Deze grammatica is niet $LL(1)$—zij is zelfs niet $LL(k)$ voor welke k dan ook. Immers voor iedere k kunnen E de expressies $(((\ldots(a)\ldots))) + a$ en $(((\ldots(a)\ldots))) - a$ met $k+1$ openingshaakjes worden gegenereerd. k stappen vooruitkijken is niet voldoende om vast te stellen welke regel moet worden gebruikt $E \to E+T$ of juist $E \to E-T$.

Maar we moeten nog niet wanhopen! Er zijn nog verschillende strategieën mogelijk om het probleem toch op te lossen. Een voor de hand liggende benadering is te zoeken naar een equivalente grammatica die wel $LL(1)$ is. In ieder geval moeten we dan de linker recursie uit de grammatica verwijderen



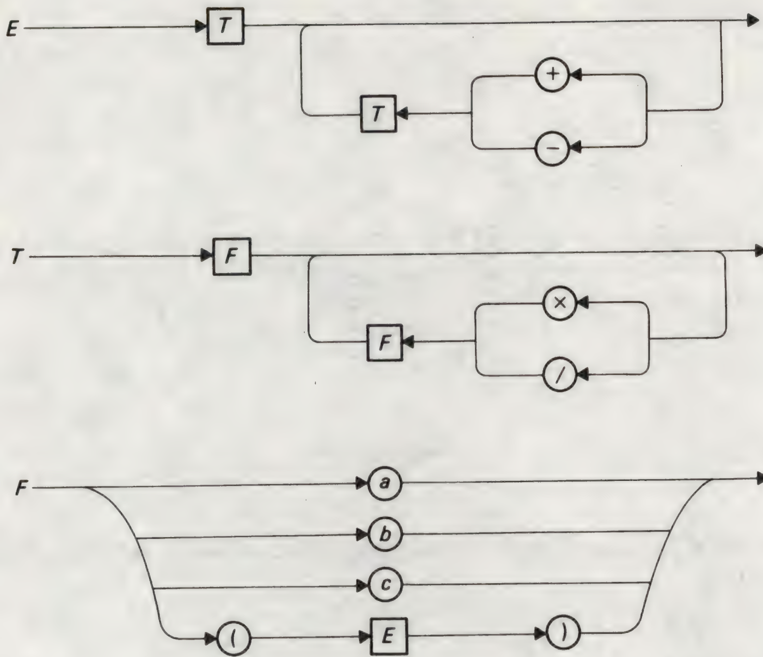
Figuur 7.3

(zie oefening 7.2). Met behulp van stelling 5.6 construeren we een equivalente grammatica

$$S \rightarrow E\$$$
$$E \rightarrow TA$$
$$A \rightarrow \varepsilon| \ +TA| \ -TA$$
$$T \rightarrow FB$$
$$B \rightarrow \varepsilon| \ \times FB| \, /FB$$
$$F \rightarrow a|b|c|(E)$$

Deze grammatica is wel $LL(1)$ en dus kunnen we er gemakkelijk een herkenner met recursieve afdaling voor ontwerpen. Toch hebben we nog pech: het invoegen van de semantische handelingen is wel mogelijk, maar het is niet erg eenvoudig. Onze oorspronkelijke grammatica was met opzet zo geformuleerd dat de semantische boom gemakkelijk uit de afleidingsboom kon worden geconstrueerd. Door de grammatica te herschrijven is deze eigenschap ten dele verloren gegaan. De string $a - b - c\$$, bijvoorbeeld heeft als afleidingsboom figuur 7.3a, maar de juiste semantische boom is die in figuur 7.3b en niet die in 7.3c.

Een andere benadering is het weergeven van de grammatica in syntaxisdiagrammen zoals in figuur 7.4. Hiermee vermijd je het gebruik van de nonterminale symbolen A en B en het blijkt dat je nu een herkenner met recursieve afdaling kunt schrijven waarin de correcte semantische handelingen gemakkelijk kunnen worden ingevoegd. Voor ieder van de nonterminalen in de syntaxisdiagrammen schrijven we een procedure. We krijgen dus procedures pE, pT en pF. De procedures $p+$, $p-$, enzovoort voor het herkennen van terminale symbolen bouwen we, zoals ook in de praktijk gebruikelijk is, in in de procedures voor de nonterminalen, voorzover en waar nodig.



Figuur 7.4

De herkenner krijgt dan de volgende vorm.

begin pE; *if volgendsymbool* = '*S*' *then* halt *else* mis *end*;

Met als procedures:

```
procedure pE ≡ begin
                 pT;
                 while volgendsymbool = '+' or volgendsymbool = '−' do
                       begin
                           read(symbool); pT
                       end
                 end;
procedure pT ≡ begin
                 pF;
                 while volgendsymbool = 'x' or volgendsymbool = '/' do
                       begin
                           read(symbool); pF
                       end
                 end;
```

```
procedure pF ≡ begin
               case volgendsymbool of
               'a', 'b', 'c' : read(symbool);
                         '(' : read(symbool); pE; read(symbool);
                               if symbool ≠ ')' then mis;
                    ')', '$' : mis
               end;
```

Nu moeten we de semantische handelingen in de procedures inbouwen. We construeren een functie fX uit iedere procedure pX, en deze functie levert een semantische boom op die de rekenkundige expressie voorstelt die door pX wordt herkend. We gaan uit van een functie '*maakboom*' met drie parameters: een symbool s en twee bomen T_1 en T_2. De functie construeert hieruit een boom met wortel s en met als linker subboom T_1 en als rechter subboom T_2. De lege boom noemen we nil.
In onderstaande pseudocode zijn de semantische acties ingevoegd.

```
begin var t: boom; t := fE; if volgendsymbool = '$' then t else mis end;
```

De functies worden:

```
function fE: boom ≡ begin
                    var t: boom;
                    t := fT;
                    while volgendsymbool = '+' or volgendsymbool = '−' do
                          begin
                             read(symbool); t := maakboom(symbool,t,fT)
                          end;
                          fE := t
                    end;
function fT: boom ≡ begin
                    var t: boom;
                    t := fF;
                    while volgendsymbool = 'x' or volgendsymbool = '/' do
                          begin
                             read(symbool); t := maakboom(symbool,t,fF)
                          end;
                          fT := t
                    end;
function fF: boom ≡ begin
                    case volgendsymbool of
                    'a', 'b', 'c' : read(symbool);
                                    fF := maakboom(symbool,nil,nil);
                              '(' : read(symbool); fF := fE; read(symbool);
                                    if symbool ≠ ')' then mis;
                         ')', '$': mis
                    end;
```

Bij compilerbouw blijkt de techniek van de recursieve afdaling heel effectief te kunnen toegepast. We toonden aan dat een herkenner voor een $LL(1)$-grammatica eenvoudig kan worden afgeleid uit de KIJKVOORUIT verzamelingen. Vervolgens kunnen de bijbehorende semantische handelingen worden ingevoegd om er een ontleder van te maken. In de praktijk kan de syntaxis van de meeste programmeertalen echter niet volledig met behulp van een $LL(1)$-grammatica worden gespecificeerd. De compilerbouwer kan wel voor een groot deel van de taal gebruik maken van de hierboven beschreven methode, maar er zullen gedeelten overblijven waarop de methode niet toepasbaar blijkt. Dit komt ofwel omdat die stukken niet in $LL(1)$-vorm kunnen worden geformuleerd, ofwel omdat dan het invoegen van semantische acties erg moeilijk wordt. Voor dit soort syntaxisregels moet de compilerschrijver dus van andere methodieken gebruik maken. Als men bijvoorbeeld k stappen vooruit kan kijken, dan kan de methode worden gegeneraliseerd tot een $LL(k)$-grammatica. Zie *Recursive Descent Compiling* door A.J.T. Davie en R. Morrison, (uitg. Ellis Horwood, 1981) voor een uitgebreide praktische toepassing van de methode op een Algol-achtige taal, (S-Algol).

In *Syntax of Programming Languages: Theory and Practice* door R.C. Backhouse, (Prentice-Hall International, 1979) wordt de theoretische behandeling van $LL(k)$-grammatica's verder uitgewerkt.

7.3 Oefeningen

1. Welke van de volgende grammatica's zijn $LL(1)$? Ontwerp voor die grammatica's bijbehorende recursieve afdaling herkenners.

   (a) $S \rightarrow A|B$
   $A \rightarrow aA|a$
   $B \rightarrow bB|b$

   (b) $S \rightarrow AB$
   $A \rightarrow Ba|\varepsilon$
   $B \rightarrow Cb|C$
   $C \rightarrow c|\varepsilon$

   (c) $S \rightarrow aAaB|bAbB$
   $A \rightarrow S|cb$
   $B \rightarrow cB|a$

2. Als we weten dat er volgens afspraak geen irrelevante produktieregels in CFG's voorkomen, bewijs dan dat een $LL(1)$-grammatica geen linksrecursieve produkties kan hebben. Geldt dit ook voor $LL(k)$-grammatica's?
3. Generaliseer stelling 7.3 en de daaraan voorafgaande definities tot een nodige en voldoende voorwaarde in termen van k-staps KIJKVOORUIT-verzamelingen voor streng $LL(k)$ zijn van een grammatica.
4. Gebruik het resultaat van oefening 7.3 om te bewijzen dat de volgende grammatica streng $LL(2)$ is en ontwerp vervolgens een recursieve afdaling

herkenner.

$$S \rightarrow aAS|AbSc|\varepsilon$$
$$A \rightarrow cbA|a$$

5. (a) Bewijs dat de volgende grammatica $LL(2)$ is maar niet streng $LL(2)$.

$$S \rightarrow aAaa|bAba$$
$$A \rightarrow b|\varepsilon$$

   (b) Bewijs dat de volgende grammatica $LL(3)$ is maar niet streng $LL(k)$ voor $k \geq 1$.

$$S \rightarrow aBA|bBbA$$
$$A \rightarrow abA|c$$
$$B \rightarrow a|ab$$

6. Als L een reguliere taal is en $\$ \notin L$ dan kan $L\$ = \{x\$|x \in L\}$ worden gegenereerd door een (strenge) $LL(1)$-grammatica. Bewijs dit. (Hint: ga uit van de deterministische eindige automaat die de taal L accepteert.)
7. Als een grammatica $G = (N, T, P, S)$ meerduidig (ambigu) is dan bestaan elementen $w, x \in T^*, A \in N, \gamma \in (N \cup T)^*$ zodanig dat $S \overset{*}{\Rightarrow} wA\gamma$ een linkerafleiding is en verder bestaan er twee verschillende afleidingen $A \Rightarrow \alpha \overset{*}{\Rightarrow} x$ en $A \Rightarrow \beta \overset{*}{\Rightarrow} x$ met $\alpha, \beta \in (N \cup T)^*$ en $\alpha \neq \beta$. Bewijs dit en gebruik dit resultaat om te bewijzen dat iedere $LL(k)$-grammatica eenduidig is.
8. Bewijs dat de grammatica met de volgende produktieregels

$$S \rightarrow C\$$$
$$C \rightarrow bA|aB$$
$$A \rightarrow a|aC|bAA$$
$$B \rightarrow b|bC|aBB$$

niet $LL(k)$ is voor enige k. Kunt u een equivalente grammatica construeren die voor zekere k wel $LL(k)$ is?

Hoofdstuk 8

Opwaartse ontleding

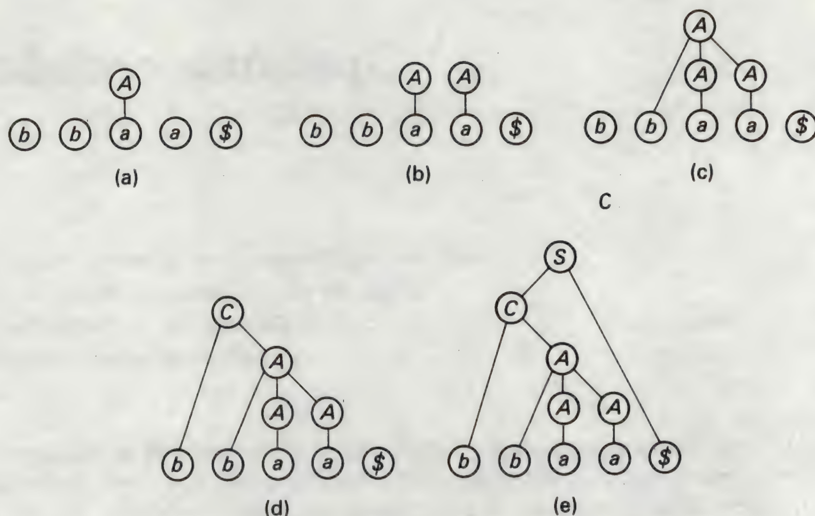
Build from the bottom up not from the top down.
Begin bij het fundament; niet bij het dak.
—FRANKLIN D. ROOSEVELT
Radiotoespraak, 7 april 1932

Als een contextvrije grammatica G eenduidig is dan moet een opwaartse ontleding van een string $x \in L(G)$ dezelfde afleidingsboom tot resultaat hebben als een neerwaartse, (voorzover deze technieken kunnen worden toegepast). De methoden verschillen alleen wat betreft de wijze waarop de bomen worden geconstrueerd. De neerwaartse (top-down) ontleedmethoden die we in het vorige hoofdstuk behandelden trachten de ontleedboom vanaf de top op te bouwen; de opwaartse (bottom-up) methoden beginnen juist met de blaadjes en werken van daaruit naar de top (de wortel) van de boom. De invoerstring wordt van links naar rechts gelezen om substrings te zoeken die overeenkomen met symbolen in het rechterlid van een produktieregel. Zodra zo'n substring is bepaald, kan deze vervangen worden door of *gereduceerd* worden tot het nonterminale symbool in het linkerlid van deze produktieregel en zo wordt een zinsvorm van de grammatica verkregen. In deze zinsvorm zoeken we weer een reduceerbare substring. Het uiteindelijke doel is het vinden van een rij reducties die het mogelijk maakt de hele afleidingsboom vanuit de blaadjes tot aan de wortel op te bouwen. Iedere reductie komt overeen met het bepalen van een ouder van een knooppunt in de boom.

Beschouw de grammatica G met produktieregels

$$S \rightarrow C\$$$
$$C \rightarrow aBC|aB|bAC|bA$$
$$A \rightarrow bAA|a$$
$$B \rightarrow aBB|b$$

Op grond van oefening 2.5 weten we al dat deze grammatica eenduidig is en dus heeft iedere $x \in L(G_1)$ een unieke afleidingsboom. De afleidingsboom voor $x = bbaa\$$ bijvoorbeeld is in figuur 8.1e getekend. Bij iedere afleidingsboom hoort een unieke *rechterafleiding*, dit is een afleiding waarin het meest rechtse nonterminale symbool het eerst verder wordt uitgewerkt. Bij de boom in figuur 8.1 hoort de rechterafleiding $S \Rightarrow C\$ \Rightarrow bA\$ \Rightarrow bbAA\$ \Rightarrow bbAa\$ \Rightarrow bbaa\$$.



Figuur 8.1

De opwaartse ontleding die we hier zullen behandelen construeert zo'n rechterafleiding, maar van achteren naar voren. De constructie van de afleidingsboom voor *bbaa$* vindt dan plaats zoals in figuur 8.1 aangegeven.

De substring die bij elke stap verder wordt gereduceerd heet het *handvat* (Engels: handle) van de rechter zinsvorm. In tabelvorm kunnen we het ontleedproces zo weergeven.

Rechter Zinsvorm	Handvat	Vervangen door
bbaa$	(eerste) *a*	*A*
bbAa$	*a*	*A*
bbAA$	*bAA*	*A*
bA$	*bA*	*C*
C$	*C$*	*S*

Het kernprobleem van de opwaartse ontleedmethode is het telkens vinden van een handvat en het op geschikte wijze reduceren daarvan bij iedere stap.

8.1 Eenvoudige precedentiegrammatica's

Het bovengenoemde probleem van het bepalen van handvatten en het reduceren daarvan behandelen we eerst voor een kleinere klasse van grammatica's, de zogenaamde eenvoudige precedentiegrammatica's.

We doorlopen de meest rechtse zinsvorm van links naar rechts en kijken naar paren naburige symbolen om de 'staart' van het handvat te vinden, dat wil zeggen het symbool uiterst rechts. Zodra we die hebben gevonden gaan we naar links om de 'kop' van het handvat te vinden, dit is het symbool uiterst links.

Zij $G = (N, T, P, S)$ een contextvrije grammatica en zij $\alpha XY\beta$ een rechter zinsvorm, met $\alpha, \beta \in (N \cup T)^*$ en $X, Y \in N \cup T$. Tijdens de reductie van deze zinsvorm naar S kan op een zeker moment één van de volgende drie situaties optreden:

(a) X is deel van het handvat, maar Y is dat niet. X is dus de staart van het handvat. We zeggen dan dat X voorgaat voor Y en schrijven $X \gtrdot Y$.
(b) X en Y zitten beiden in het handvat. X en Y hebben dan dezelfde waarde en we schrijven $X \doteq Y$.
(c) Y zit in het handvat maar X niet. Dan is Y de kop van het handvat en dan gaat Y voor voor X; notatie X$\lessdot$Y.

De relaties $\gtrdot$, $\doteq$ en $\lessdot$ heten eenvoudige precedentierelaties. Uit de definities volgt dat

(a) $X \gtrdot Y$ desd als Y een terminaal symbool is (bedenk dat $\alpha XY\beta$ een zinsvorm is in een rechterafleiding) en als er een produktieregel $A \to \gamma_1 BZ\gamma_2$ bestaat met $A, B \in N, Z \in N \cup T, \gamma_1, \gamma_2 \in (N \cup T)^*$ zo dat $B \stackrel{+}{\Rightarrow} \delta_1 X$ en $Z \stackrel{*}{\Rightarrow} Y\delta_2$ met $\delta_1, \delta_2 \in (N \cup T)^*$.
(b) $X \doteq Y$ desd als er een produktie $A \to \gamma_1 XY\gamma_2$ bestaat met $A \in N$ en $\gamma_1, \gamma_2 \in (N \cup T)^*$.
(c) $X \lessdot Y$ desd als er een produktie $A \to \gamma_1 XB\gamma_2$ bestaat met $B \in N$, $\gamma_1, \gamma_2 \in (N \cup T)^*$ zodanig dat $B \stackrel{+}{\Rightarrow} Y\delta, \delta \in (N \cup T)^*$.

Een contextvrije grammatica $G = (N, T, P, S)$ heet een eenvoudige precedentiegrammatica onder de volgende voorwaarden:

(i) geen twee regels van de grammatica hebben identieke rechterleden
(ii) er bestaat hoogstens één eenvoudige precedentierelatie tussen een paar symbolen in $(N \cup T) \times (N \cup T)$.

G_1 voldoet weliswaar aan de eerste voorwaarde, maar niet aan de tweede omdat zowel $a \gtrdot a$ (wegens $A \to bAA, A \stackrel{+}{\Rightarrow} a$ en $A \stackrel{*}{\Rightarrow} a$) alsook $a \lessdot a$ (wegens $C \to aB$ en $B \stackrel{+}{\Rightarrow} aBB$).

Beschouw nu de eenvoudige precedentiegrammatica G met regels

$$S \to (R|a$$
$$R \to Sa)$$

Ga na dat tabel 8.1 de bijbehorende eenvoudige precedentierelaties bevat.

Als een grammatica $G = (N, T, P, S)$ een eenvoudige precedentiegrammatica is dan kan het handvat van iedere rechter zinsvorm gemakkelijk worden

	S	R	a	(	)
S	·	·	$\doteq$	·	·
R	·	·	$\gtrdot$	·	·
a	·	·	$\gtrdot$	·	$\doteq$
(	$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	·
)	·	·	$\gtrdot$	·	·

Tabel 8.1

gevonden. We doorlopen eenvoudig de zinsvorm en zoeken naar het meest linker paar symbolen X_j en X_{j+1} met $X_j \gtrdot X_{j+1}$ · X_j is dan de staart van het handvat. Vervolgens doorlopen we de zinsvorm van rechts naar links, beginnend bij X_j totat we een paar X_{i-1}, X_i vinden met $X_{i-1} \lessdot X_i$ · X_i is dan de kop van het handvat. Als we het handvat $\beta = X_i \dots X_j$ eenmaal hebben gevonden dan moet er een unieke produktieregel van de vorm $A \rightarrow \beta$ bij bestaan en dus kunnen we reduceren tot A. Alleen als de staart van het handvat juist het laatste symbool is of als de kop het eerste symbool is dan ontstaat er een probleem. Om dit op te lossen introduceren we een eindesymbool $\$ \notin N \cup T$ dan we aan beide einden van onze zinsvormen plaatsen. We stellen $\$ \gtrdot X$ en $X \gtrdot \$$ voor alle $X \in N \cup T$ en door het herhaald reduceren van handvatten brengen we de inputstring $\$x\$, x \in L(G)$ terug tot $\$S\$$.

$L(G_2)$ bevat strings van de vorm a, (aa), $((aa)a)$, $(((aa)a)a)$ enzovoort. In tabel 8.2 laten we zien hoe $\$(((aa)a)a)\$$ wordt ontleed met behulp van de precedentierelaties. De bijbehorende rechterafleiding is $S \Rightarrow (R \Rightarrow (Sa) \Rightarrow ((Ra) \Rightarrow ((Sa)a) \Rightarrow (((Ra)a) \Rightarrow (((Sa)a)A) \Rightarrow (((aa)a)a)$.

Een handige methode om de reductie van handvatten uit te voeren maakt gebruik van een stack en een inputbuffer. De belangrijkste bewerkingen die worden gebruikt zijn de push, (het plaatsen van een element op de stack) en de reductie, dat wil zeggen het vervangen van het handvat als deze zich als top-element op de stack bevindt. De ontleder accepteert input die eindigt met $\$S$ op de stack en met $\$$ in het inputbuffer. Als geen volgende stap mogelijk blijkt dan wordt een foutprocedure aangeroepen. Dit soort ontleders worden wel *schuif-reduceerontleders* (Engels: shift-reduce parser) genoemd. In tabel 8.3 zijn de inhouden van stack en inputbuffer na iedere stap toegepast op ons voorbeeld, weergegeven.

Bij de uitwerking van een schuif-reduceerontleder wordt steeds een symbool op de stack geplaatst als het laatste op de stack geplaatste symbool een precedentie heeft lager dan of gelijk aan het volgende in het inputbuffer. Als dit niet zo is dan worden symbolen van de stack gehaald totdat het handvat gevonden is en een reductie kan worden uitgevoerd. Wordt er geen handvat gevonden dan wordt de foutprocedure aangeroepen.

Om een ontleder volgens de eenvoudige precedentiemethode te bouwen moeten eerst de precedentierelaties worden vastgesteld. Gelukkig bestaat hiervoor

Zinsvorm	Handvat
$\$\ (\ (\ (\ a\ \ a)a\ )a)\ \$$	a
$\lessdot\ \lessdot\ \lessdot\ \lessdot\ \gtrdot$	
$\$\ (\ (\ (\ S\ a\ )\ \ a)a)\$$	$Sa)$
$\lessdot\ \lessdot\ \lessdot\ \lessdot\ \doteq\ \doteq\ \gtrdot$	
$\$\ (\ (\ (\ R\ a)a\ )\$$	$(R$
$\lessdot\ \lessdot\ \lessdot\ \doteq\ \gtrdot$	
$\$\ (\ (\ S\ a\ )\ \ a)\ \$$	$Sa)$
$\lessdot\ \lessdot\ \lessdot\ \doteq\ \doteq\ \gtrdot$	
$\$\ (\ (\ R\ a)\$$	$(R$
$\lessdot\ \lessdot\ \doteq\ \gtrdot$	
$\$\ (\ S\ a\ )\ \$$	$Sa)$
$\lessdot\ \lessdot\ \doteq\ \doteq\ \gtrdot$	
$\$\ (\ R\ \$$	$(R$
$\lessdot\ \doteq\ \gtrdot$	
$\$\ S\ \$$	

Tabel 8.2

een eenvoudig algoritme (zie oefening 8.1). Alleen moet men zorgvuldig te werk gaan om opslagruimteproblemen te voorkomen. Immers als de grammatica m terminale en n nonterminale symbolen heeft dan zijn er $(m+n)^2$ mogelijke relaties. Als we die weergeven door een precedentiematrix dan zullen daarvan veel elementen de waarde $\cdot$ hebben (geen relatie). Beter is de matrix te comprimeren met een van de vele technieken voor het opslaan van 'schaarse' arrays (rijen met veel nullen). Een andere mogelijkheid is het gebruik van precedentiefuncties. Uit een precedentiematrix P trachten we twee geheeltallige functies f en g te construeren met

$$f(i) < g(j) \text{ als } P_{ij} \equiv \lessdot,$$
$$f(i) = g(j) \text{ als } P_{ij} \equiv \doteq$$

en

$$f(i) > g(j) \text{ als } P_{ij} \equiv \gtrdot.$$

Deze functies heten, als we ze kunnen vinden, precedentiefuncties en kunnen worden opgeslagen in $2(m+n)$ geheugenposities, in plaats van in $(m+n)^2$ zoals de precedentiematrix zelf. Hiermee is wel informatie verloren gegaan, namelijk het afwezig zijn van een precedentierelatie. Deze informatie kan nuttig zijn in verband met het afvangen van fouten.

Veel fundamenteler is de kritiek dat deze methode niet vaak kan worden toegepast omdat maar weinig grammatica's eenvoudige precedentiegrammatica's zijn. Het gaat hier slechts om een kleine subklasse van de klasse der contextvrije grammatica's.

Stack	Inputbuffer	Actie
$\$$	$(((aa)a)a)\$$	schuif (plaats op de stack)
$\$($	$((aa)a)a)\$$	schuif
$\$(($	$(aa)a)a)\$$	schuif
$\$((($	$aa)a)a)\$$	schuif
$\$(((a$	$a)a)a)\$$	reduceer met $S \rightarrow a$
$\$(((S$	$a)a)a)\$$	schuif
$\$(((Sa$	$)a)a)\$$	schuif
$\$(((Sa)$	$a)a)\$$	reduceer met $R \rightarrow Sa)$
$\$(((R$	$a)a)\$$	reduceer met $S \rightarrow (R$
$\$((S$	$a)a)\$$	schuif
$\$((Sa$	$)a)\$$	schuif
$\$((Sa)$	$a)\$$	reduceer met $R \rightarrow Sa)$
$\$((R$	$a)\$$	reduceer met $S \rightarrow (R$
$\$(S$	$a)\$$	schuif
$\$(Sa$	$)\$$	schuif
$\$(Sa)$	$\$$	reduceer met $R \rightarrow Sa)$
$\$(R$	$\$$	reduceer met $S \rightarrow (R$
$\$S$	$\$$	accepteer

Tabel 8.3

8.2 LR(0)-Grammatica's

De opwaartse ontleedmethode die meestal in de praktijk wordt gebruikt is de LR-methode die is ontwikkeld door Donald Knuth, (Knuth, D.E. (1965) 'On the translation of languages from left to right', *Information and Control*, **8**, 607-639). De L in LR staat voor 'Lees van Links naar rechts' en de R voor 'construeer een omgekeerde Rechterafleiding'. De LR-methode is net als de eenvoudige precedentiemethode een schuif-reduceerontleder. De LR-methode is echter veel algemener toepasbaar en daarom te prefereren. In feite is LR zelfs beter dan LL, omdat LR op een grotere klasse grammatica's kan worden toegepast. Een nadeel van LR is dat de methode niet zo intuïtief begrijpelijk is als LL, maar eenmaal uitgewerkt en geprogrammeerd blijkt het desalniettemin een goed werkende methode.

Bij een schuif-reduceerontleder, zoals weergegeven in tabel 8.3 heeft de stack op ieder tijdstip een inhoud $\alpha \in (N \cup T)^*$ en heeft het inputbuffer een inhoud $z \in T^*$. Samen vormen αz een rechter zinsvorm. Bij iedere stap kunnen we ofwel een schuif-operatie uitvoeren, (het plaatsen van een volgend symbool uit de inputbuffer op de stack), ofwel we kunnen een reductie uitvoeren. Het probleem is steeds te bepalen welke van de twee operaties moet worden toegepast en, als het een reductie is, welke produktieregel dan moet worden gebruikt. Het liefst willen we zodanige voorwaarden aan de grammatica opleggen dan we altijd vanuit onze kennis van de inhoud van de stack precies kunnen bepalen welke

volgende stap onze schuif-reduceerontleder moet uitvoeren om uiteindelijk de omkering van een rechterafleiding te produceren.

Als we als produktieregel $A \rightarrow \beta$ hebben en de stackinhoud is $\alpha = \gamma\beta$, dan kunnen we een reductie uitvoeren. Dit moet de enig mogelijke stap zijn, dat wil zeggen het moet onmogelijk zijn om nul of meer schuifoperaties uit te voeren die de stackinhoud op $\alpha x, x \in T^*$ zetten om vervolgens een of andere reductie $B \rightarrow \beta'$ toe te passen en zo ook een rechterzinsvorm te bereiken. We eisen dus als er twee rechterafleidingen zijn

$$S \overset{*}{\Rightarrow} \gamma A x y \Rightarrow \gamma\beta x y$$

en

$$S \overset{*}{\Rightarrow} \delta B y \Rightarrow \delta\beta' y = \gamma\beta x y$$

met $\gamma, \delta, \beta, \beta' \in (N \cup T)^*, A, B, \in N$ en $x, y \in T^*$, dan moeten deze twee afleidingen hetzelfde zijn, dus $\delta = \gamma, A = B, \beta = \beta'$ en $x = \varepsilon$.

De verzameling strings die op de stack terecht kan komen voordat er een reductie kan worden uitgevoerd met een produktieregel $A \rightarrow \beta$ heet de LRCONTEXT-verzameling van $A \rightarrow \beta$; notatie LRCONTEXT($A \rightarrow \beta$). Formeel,

$$\text{LRCONTEXT}(A \rightarrow \beta) = \{\alpha | \alpha = \gamma\beta \in (N \cup T)^* \text{ waarbij } S \overset{*}{\Rightarrow} \gamma A x \Rightarrow \gamma\beta x \text{ een rechterafleiding is met } x \in T^* \text{ en } \gamma \in (N \cup T)^*\}.$$

Aan bovenstaande voorwaarde is niet voldaan wanneer er verschillende produkties $A \rightarrow \beta$ en $B \rightarrow \beta'$ bestaan zodanig dat LRCONTEXT($A \rightarrow \beta$) $\gamma\beta$ bevat en LRCONTEXT($B \rightarrow \beta'$) $\delta\beta'$ bevat, waarbij $\delta\beta' = \gamma\beta x$ voor een $x \in T^*$.

Zelfs als een grammatica aan deze voorwaarden voldoet dan nog kunnen er problemen ontstaan over de vraag wanneer men een string moet accepteren en moet stoppen. Neem bijvoorbeeld

$$S \rightarrow Sa|a$$

LRCONTEXT($S \rightarrow a$) = $\{a\}$ en LRCONTEXT($S \rightarrow Sa$) = $\{Sa\}$, dus de regel voldoet aan de bovengenoemde voorwaarden, maar als S het enige symbool op de stack is dan is het mogelijk dat de ontleding klaar is. We weten dus niet of we moeten stoppen of dat we een schuifoperatie moeten uitvoeren. Dit probleem is echter gemakkelijk op te lossen met behulp van het einde-inputteken $\$$ en met de vermeerderde grammatica

$$S' \rightarrow S\$$$
$$S \rightarrow Sa|a.$$

Nu is LRCONTEXT($S' \rightarrow S\$$) = $\{S\$\}$, LRCONTEXT($S \rightarrow Sa$) = $\{Sa\}$ en LRCONTEXT($S \rightarrow a$) = $\{a\}$.

We formuleren op grond van het bovenstaande de volgende definitie: een contextvrije grammatica $G = (N, T, P, S)$ is een $LR(0)$-grammatica als

(a) het startsymbool niet in het rechterlid van de produktieregels voorkomt
(b) er twee rechterafleidingen zijn

$$S \overset{*}{\Rightarrow} \gamma Axy \Rightarrow \gamma\beta xy$$

en

$$S \overset{*}{\Rightarrow} \delta By \Rightarrow \delta\beta' y = \gamma\beta xy$$

met $\gamma, \delta, \beta, \beta' \in (N \cup T)^*$, $A, B \in N$ en $x, y \in T^*$, dan geldt $d = \gamma, A = B, \beta = \beta'$ en $x = \varepsilon$.

Uit deze definitie volgt ook de volgende stelling.

Stelling 8.1

Een contextvrije grammatica $G = (N, T, P, S)$ is een $LR(0)$-grammatica dan en slechts dan als

(a) het startsymbool niet voorkomt in het rechterlid van een van de produktieregels,
(b) als $\alpha \in$ LRCONTEXT($A \rightarrow \beta'$) en $\alpha x \in$ LRCONTEXT($B \rightarrow \beta'$) met $A \rightarrow \beta, B \rightarrow \beta' \in P$, $\alpha \in (N \cup T)^*$ en $x \in T^*$ dan geldt $x = \varepsilon, A = B$ en $\beta = \beta'$.

Om na te gaan of een contextvrije grammatica, die voldoet aan de eerste voorwaarde in deze stelling, een $LR(0)$-grammatica is, moeten we de LRCONTEXT verzamelingen berekenen. Iedere string in LRCONTEXT ($A \rightarrow \beta$) is van de vorm $\gamma\beta$ voor een of andere $\gamma \in (N \cup T)^*$. Hieruit volgt de volgende stelling als we voor alle $A \in N$ definiëren: LINKS(A) $= \{\gamma | S \overset{*}{\Rightarrow} \gamma Ax$ is een rechterafleiding, $x \in T^*\}$.

Stelling 8.2

LRCONTEXT($A \rightarrow \beta$) = LINKS(A) $\cdot \{\beta\}$.

Om dus de LRCONTEXT-verzamelingen te vinden van de produktieregels van de grammatica, moeten we eerst de LINKS-verzamelingen bepalen van de nonterminale symbolen. Als we aannemen dat in de grammatica $G = (N, T, P, S)$ S niet voorkomt in de rechterleden van de produktieregels dan volgt hieruit dat LINKS(S) $= \{\varepsilon\}$. Ook geldt als $B \rightarrow \gamma_1 A \gamma_2$ een produktieregel van G is, dat LINKS(A) dan LINKS(B) $\cdot \{\gamma_1\}$ bevat.

We bekijken nu de grammatica G_3 met als produktieregels

$$S \rightarrow A\$$$
$$A \rightarrow AB|B$$
$$B \rightarrow [A]\,|[\,]$$

om na te gaan hoe we de LRCONTEXT-verzamelingen kunnen bepalen. De grammatica genereert strings met correcte paren linker en rechter vierkante haakjes en met $\$$ als eindsymbool. Voorbeelden: []$,[[]]$, [][]$, enzovoort.

Als we de hierboven afgeleide regels over de LINKS-verzamelingen toepassen dan krijgen we de volgende vergelijkingen

$$\begin{aligned}
&\text{LINKS}(S) = \{\varepsilon\}\\
&\text{LINKS}(A) = \text{LINKS}(S) \cup \text{LINKS}(A) \cup \text{LINKS}(B) \cdot \{[\}\\
&\text{LINKS}(B) = \text{LINKS}(A)\{A\} \cup \text{LINKS}(A)
\end{aligned}$$

Deze vergelijkingen kunnen we oplossen door een omkering van de techniek die we in hoofdstuk 5 toepasten, dus door de vergelijkingen als een grammatica te herschrijven. Aan elke verzameling wijzen we een nonterminaal symbool toe: $\hat{S}$ voor LINKS(S), $\hat{A}$ voor LINKS(A) en $\hat{B}$ voor LINKS(B). De hieruit resulterende grammatica heeft als verzameling nonterminalen $\{\hat{S}, \hat{A}, \hat{B}\}$, als terminalen $\{A, [\}$ en als produktieregels

$$\begin{aligned}
&\hat{S} \rightarrow \varepsilon\\
&\hat{A} \rightarrow \hat{S} | \hat{B}[\\
&\hat{B} \rightarrow \hat{A}A | \hat{A}
\end{aligned}$$

In het algemeen kunnen de LINKS-verzamelingen van een grammatica $G = (N, T, P, S)$ worden afgeleid uit een grammatica $G = (\hat{N}, \hat{T}, \hat{P}, \hat{S})$ met $\hat{N} = \{\hat{A} | A \in N\}$ en $\hat{T} \subset N \cup T$. Uit de constructie volgt dat de produktieregels in $\hat{P}$ zodanig zullen zijn dat de grammatica linkslineair is en dus zijn de LINKS-verzamelingen allen regulier (zie oefening 3.11). Een gevolg van stelling 8.2 is dus de volgende stelling.

Stelling 8.3

Voor iedere contextvrije grammatica $G = (N, T, P, S)$ zijn de LRCONTEXT-verzamelingen van produkties in P allen regulier.

Dit resultaat biedt interessante perspectieven! Reguliere verzamelingen zijn gemakkelijk te verwerken en mogelijk kunnen we hiervan gebruik maken bij het ontwerp van een ontleder.

De constructie van een eindige automaat $\hat{M}$, die behoort bij een linkslineaire grammatica $\hat{G}$ kunnen we als volgt samenvatten. Bij iedere produktieregel $\hat{A} \rightarrow \hat{B}x, \hat{A}, \hat{B} \in N, x \in \hat{T}^*$ behoort een pad met label x vanuit een knoop $\hat{B}$ naar een knoop $\hat{A}$ en bij iedere produktieregel $\hat{A} \rightarrow x, \hat{A} \in N, x \in \hat{T}^*$ behoort een pad x vanuit de begintoestand van $\hat{M}$ (met label 1) naar knoop $\hat{A}$. Elk pad kan een of meer kanten bevatten die gelabeld zijn met elementen uit $\hat{T} \cup \{\varepsilon\}$. Als een pad uit meer dan een kant bestaat dan voeren we tussenknopen in die we met 2, 3, 4, enzovoort, labelen. Als we nu de overgangsfunctie van $\hat{M}$, $\hat{t}$ noemen, dan volgt uit onze constructie dat voor alle $A \in N$ geldt dat LINKS(A) $= \{x \in \hat{T}^* | \hat{t}(1, x) = \hat{A}\}$.

In figuur 8.2a is de eindige automaat die hoort bij dit voorbeeld getekend.

Nu gelden de volgende gelijkheden:

$$\begin{aligned} \text{LINKS}(S) &= 1 \\ \text{LINKS}(A) &= ((A+1)[\,])^* \\ \text{LINKS}(B) &= ((A+1)[\,])^*(A+1) \end{aligned}$$

en dus volgt

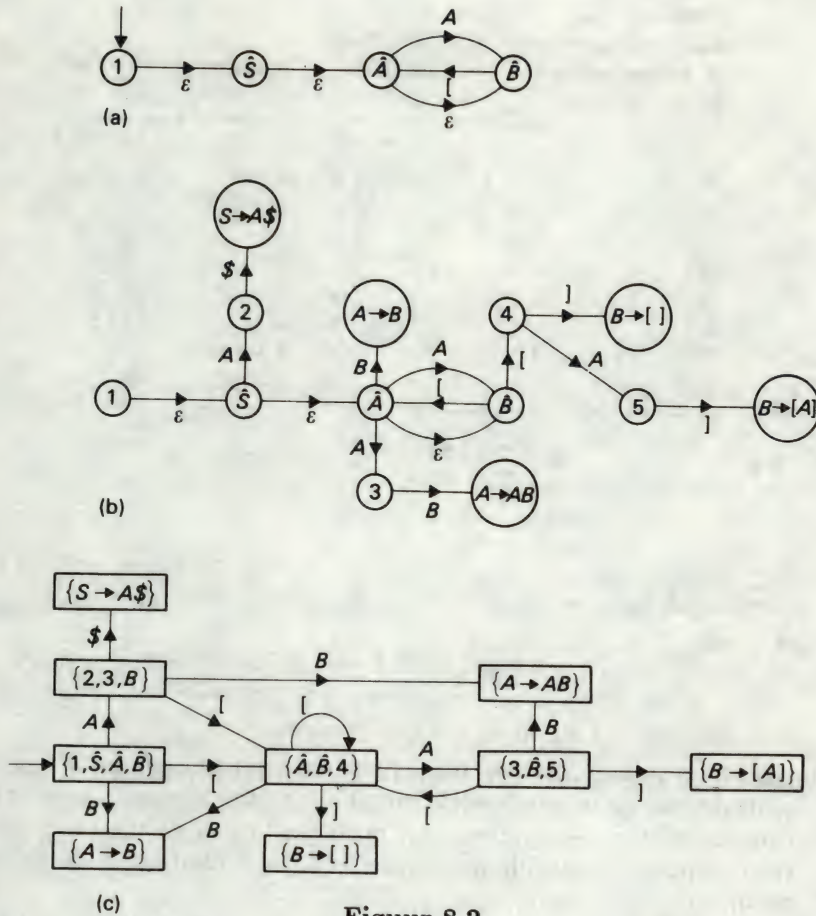
$$\begin{aligned} \text{LRCONTEXT}(S \rightarrow A\$) &= A\$ \\ \text{LRCONTEXT}(A \rightarrow AB) &= ((A+1)[\,])^* AB \\ \text{LRCONTEXT}(A \rightarrow B) &= ((A+1)[\,])^* B \\ \text{LRCONTEXT}(B \rightarrow [A]) &= ((A+1)[\,])^*(A+1)[A] \\ \text{LRCONTEXT}(B \rightarrow [\,]) &= ((A+1)[\,])^*(A=1)[\,] \end{aligned}$$

Op grond van stelling 8.1 is nu in te zien dat G_3 inderdaad $LR(0)$ is.

Als we de eindige automaat $\hat{M}$ hebben geconstrueerd dan kunnen we vervolgens stelling 8.2 gebruiken voor de constructie van een tweede automaat die een toestand heeft bij iedere produktieregel uit de oorspronkelijke grammatica en die een zodanige structuur heeft dat de verzameling strings die ons van een begintoestand naar een toestand brengen met als label een van de produktieregels, juist overeenkomt met de LRCONTEXT-verzameling voor die produktieregel. Bij ons voorbeeld leidt dit tot de automaat die in figuur 8.2b is weergegeven. In figuur 8.2c is de equivalente deterministische automaat zonder ε-produkties getekend. Deze laatste automaat wordt de *karakteristieke automaat* bij de grammatica genoemd. De knopen van een karakteristieke automaat die een produktieregel als label hebben, heten *reductietoestanden*. Een grammatica is $LR(0)$ dan en slechts dan als de reductietoestanden van de bijbehorende karakteristieke automaat elk met precies een produktieregel zijn gelabeld en als zij geen terminale symbolen bevatten.

We laten nu zien hoe de karakteristieke automaat uit ons voorbeeld kan worden gebruikt om na te gaan dat de string $[[\,][\,]][\,]\$$ tot de taal behoort. We voeren de string symbool voor symbool in totdat we in een reductietoestand terechtkomen. Nu weten we welke reductie we moeten toepassen en zo krijgen we een nieuwe (rechter) zinsvorm. Nu kunnen we opnieuw vaststellen welke reductie we moeten toepassen door een prefix van deze zinsvorm in te voeren in de karakteristieke automaat. Als we dit herhaald toepassen dan worden de stappen uitgevoerd die zijn aangegeven in tabel 8.4.

We gebruiken de automaat zo niet erg efficiënt. Het steeds opnieuw opstarten van de automaat na een reductie is in feite niet nodig. In stap 4 bijvoorbeeld hebben we zojuist $[\,]$ tot B gereduceerd. Als we tijdens de vorige stap hadden onthouden dat we ons in toestand $\{3, \hat{B}, 5\}$ bevonden, na de verwerking van $[A$ en voor de verwerking van $[\,]$, dan hadden we geweten dat de input $[AB$ zou leiden tot toestand $t(\{3, \hat{B}, 5\}, B)$. Voordat we dit idee verder uitwerken, vereenvoudigen we eerst onze werkwijze door de toestanden te nummeren met 1, 2, ... Hierbij is 1 de begintoestand en in een *reductietabel* houden we bij welke



Figuur 8.2

toestanden reductietoestanden zijn. Als toestand k een reductietoestand is met bijvoorbeeld $A \longrightarrow \beta$ als bijbehorende produktieregel dan zetten we $A \longrightarrow \beta$ op de k^{de} regel van de reductietabel. In figuur 8.3 geven we het resultaat weer van het opnieuw labelen te zamen met de bijbehorende reductietabel.

Nu is het mogelijk een efficiënt ontleedalgoritme te formuleren met behulp van een toestandstack. Bij iedere stap van het algoritme bevat de stack de toestanden die de karakteristieke automaat zou hebben doorlopen als we de huidige prefix van de rechterzinsvorm hadden ingevoerd. Als de huidige prefix bijvoorbeeld $[A[]$ is dan bevat de stack 1, 2, 3, 2, 9, waarbij element 9 de top van de stack is. Er zijn nu vier stappen mogelijk:

(a) een *reductiestap*: Stel de stackinhoud is gelijk aan $1 = k_0, k_1, \ldots, k_n = k$. Als het topelement k een reductietoestand is met als produktieregel

Stap	Prefix van de zinsvorm ingevoerd in de karakteristieke automaat	Overige invoer = restant van de zinsvorm	Toegepaste reductie	Nieuwe zinsvorm
1	[[]	[]][]$	$B \to []$	[B[]][]$
2	[B	[]][]$	$A \to B$	[A[]][]$
3	[A[]	][]$	$B \to []$	[AB][]$
4	[AB	][]$	$A \to AB$	[A][]$
5	[A]	[]$	$B \to [A]$	B[]$
6	B	[]$	$A \to B$	A[]$
7	A[]	$	$B \to []$	AB$
8	AB	$	$A \to AB$	A$
9	A$	$	$S \to AS$	S

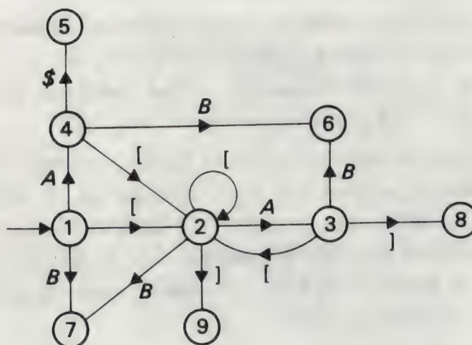
Tabel 8.4

$A \to X_1X_2 \dots X_m$ ($m \leq n$) terwijl $A \neq S$, dan gebruiken we deze produktieregel, halen m elementen van de stack en plaatsen $t(k_{n-m}, A)$ op de stack.

(b) een *schuifstap*: Als het topelement k geen reductietoestand is dan schuiven we de string op door bijvoorbeeld $a \in T$ te lezen uit de inputrij en door toestand $t(k, a)$ op de stack te plaatsen.

(c) een *acceptatiestap*: Als het topelement een reductietoestand is die overeenkomt met de unieke produktieregel die S in het linkerlid heeft staan, (bijvoorbeeld $S \to \alpha$), dan stopt het algoritme in de toestand 'accepteren', onder voorwaarde dat er precies $|\alpha| + 1$ toestanden op de stack staan.

(d) een *verwerpstap*: Als geen van de vorige drie stappen mogelijk is dan zit



Reductietabel	
5	$S \to A\$$
6	$A \to AB$
7	$A \to B$
8	$A \to [A]$
9	$B \to []$

Figuur 8.3

Stackinhoud	Stap
1	schuif [
1, 2	schuif [
1, 2, 2	schuif]
1, 2, 2, 9	reduceer met $B \rightarrow []$
1, 2, 7	reduceer met $A \rightarrow B$
1, 2, 3	schuif [
1, 2, 3, 2	schuif]
1, 2, 3, 2, 9	reduceer met $B \rightarrow []$
1, 2, 3, 6	reduceer met $A \rightarrow AB$
1, 2, 3	schuif]
1, 2, 3, 8	reduceer met $B \rightarrow [A]$
1, 7	reduceer met $A \rightarrow B$
1, 4	schuif [
1, 4, 2	schuif]
1, 4, 2, 9	reduceer met $B \rightarrow []$
1, 4, 6	reduceer met $A \rightarrow AB$
1, 4	schuif $\$$
1, 4, 5	accepteer

Tabel 8.5

er een fout in de invoer en dan geldt ofwel dat het algoritme stopt, ofwel dat een of andere poging de fout te corrigeren wordt uitgevoerd.

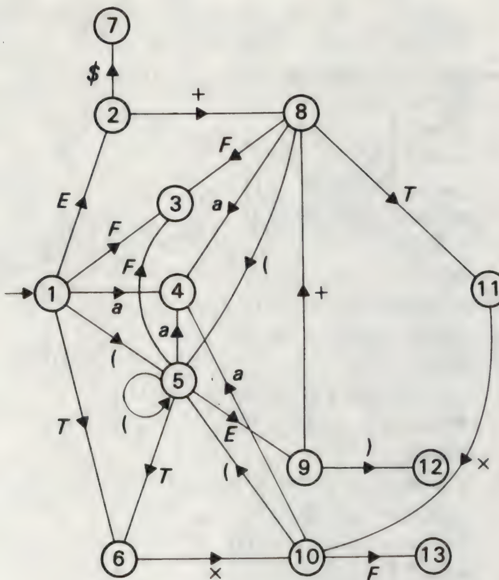
Als we de automaat met de opnieuw genummerde toestanden gebruiken dan staat in tabel 8.5 de stackinhouden tijdens de ontleding van de string [[][]][]$\$$.

Als een grammatica $LR(0)$ is dan kan de constructie van de bijbehorende karakteristieke automaat worden geautomatiseerd. Is dat eenmaal gebeurd dan is het vrij eenvoudig om een ontleder voor de taal te schrijven. Jammer genoeg komt het zelden voor dat een taal aan de $LR(0)$-voorwaarde voldoet, maar gelukkig kan de LR-ontleding op een aantal verschillende manieren worden gegeneraliseerd tot zeer efficiënte ontleedtechnieken.

8.3 LR(1)-Grammatica's

Gegeven is de grammatica G_4 met als produktieregels

$$S \rightarrow E\$$$
$$E \rightarrow E + T | T$$
$$T \rightarrow T \times F | F$$
$$F \rightarrow a | (E)$$



Reductietabel	
3	$T \rightarrow F$
4	$F \rightarrow a$
6	$E \rightarrow T$
7	$S \rightarrow E\$$
11	$E \rightarrow E + T$
12	$F \rightarrow (E)$
13	$T \rightarrow T \times F$

Figuur 8.4

In figuur 8.4 is de karakteristieke automaat getekend voor G_4. Uit deze automaat valt af te leiden dat G_4 niet $LR(0)$ is. Er geldt bijvoorbeeld $T \in$ LRCONTEXT($E \rightarrow T$), maar $T \times a \in$ LRCONTEXT($F \rightarrow a$) en $T \times F \in$ LRCONTEXT($T \rightarrow T \times F$). Als nu de huidige stackinhoud bijvoorbeeld 1, 6 is dan weten we niet of we een schuif- of een reductiestap moeten uitvoeren. Dit dilemma kunnen we in dit geval oplossen door een symbool vooruit te kijken. Als het volgende invoersymbool een × is dan moeten we een schuifstap uitvoeren, anders een reductiestap. G_4 is een voorbeeld van een zogenaamde $LR(1)$-grammatica—bij deze grammatica's kunnen we de beschreven LR-ontleder zo aanpassen dat dilemma's binnen de karakteristieke automaat worden opgelost door een 'lookahead' van één symbool. Algemeen geldt voor een $LR(k)$-grammatica ($k \geq 0$ en geheel) dat een oplossing mogelijk is door k symbolen vooruit te kijken. In de praktijk blijkt het geval $k = 1$ veruit het belangrijkste en hiertoe zullen wij ons in het navolgende beperken. Een generalisatie naar hogere k kan vrij eenvoudig worden uitgevoerd.

We nemen aan dat de wilekeurige grammatica $G = (N, T, P, S)$ al op de gebruikelijke manier is vermeerderd door het toevoegen van een speciaal eindeinvoerteken $\$$. De enige produktieregel in P die S bevat is van de vorm $S \rightarrow E\$$ voor een $E \in N$. We definiëren de LR(1)CONTEXT-verzameling van een produktieregel $A \rightarrow \beta$ in P door

$$\text{LR(1)CONTEXT}(A \rightarrow \beta) = \{\alpha | \alpha = \gamma\beta a \in (N \cup T)^*T \text{ waarbij } S \stackrel{*}{\Rightarrow} \gamma A a x \Rightarrow \gamma\beta a x \text{ een rechterproduktie is voor een } a \in T, x \in T^*, \gamma \in (N \cup T)^*\}.$$

Als $\alpha \in$ LR(1)CONTEXT($A \rightarrow \beta$) dan geldt $\alpha = \alpha' a$ voor een $\alpha \in$ LRCONTEXT($A \rightarrow \beta$) en $a \in T$. Deze $a \in T$ is de lookahead van een symbool die we in de ontleder willen opnemen. Door middel van een generalisatie van stelling 8.1 komen we tot de volgende definitie.

Als $G = (N, T, P, S)$ en vermeerderde contextvrije grammatica is dan heet G een $LR(1)$-grammatica dan en slechts dan als

> uit $\alpha \in$ LR(1)CONTEXT($A \rightarrow \beta$) en $\alpha x \in$ LR(1)CONTEXT($B \rightarrow \beta'$) met $A \rightarrow \beta, B \rightarrow \beta' \in P, \alpha \in (N \cup T)^*$ en $x \in T^*$ volgt dat $x = \varepsilon$, $A = B$ en $\beta = \beta'$.

Om te testen of een grammatica al of niet $LR(1)$ is moeten we dus de LR(1)-CONTEXT-verzamelingen berekenen van alle produktieregels. Als $A \in N$ en $a \in$ VOLGENDE(A), dan definiëren we

$$\text{LINKS}_1(A, a) = \{\gamma | S \overset{*}{\Rightarrow} \gamma A a x \text{ is een rechterafleiding, } x \in T^*\}.$$

Hieruit volgt dan

$$\text{LR(1)CONTEXT}(A \rightarrow \beta) = \bigcup_{a \in \text{VOLGENDE}(A)} \text{LINKS}_1(A, a) \cdot \{\beta a\}.$$

We moeten nu alle LINKS$_1$-verzamelingen bepalen. Dat kan op een soortgelijke manier als waarop we de LINKS-verzamelingen bepaalden. Allereerst stellen we vast dat als $B \rightarrow \gamma_1 A \gamma_2$ een produktieregel van G is met $b \in$ VOLGENDE(B) en $a \in \{\text{EERSTE}(X) | \gamma_2 b \overset{*}{\Rightarrow} x\}$ dan geldt zeker $a \in$ VOLGENDE(A) en LINKS$_1(A, a) \supset$ LINKS$_1(B, b) \cdot \{\gamma_1\}$. Zo krijgen we een stelsel vergelijkingen voor de LINKS$_1$-verzamelingen en net als bij de LINKS-verzamelingen komt dit stelsel overeen met een links-lineaire grammatica. De LINKS$_1$-verzamelingen zijn allen regulier en de LR(1)CONTEXT-verzamelingen zijn dat dus ook. Uit de grammatica die de LINKS$_1$-verzamelingen bepaalt kunnen we een deterministische eindige automaat construeren die de LR(1)CONTEXT-verzamelingen definieert. Net als bij de karakteristieke automaat voor $LR(0)$-grammatica's kunnen we bij een $LR(1)$-grammatica deze eindige automaat gebruiken om een opwaartse ontleder te construeren. Helaas heeft deze automaat meestal een groot aantal toestanden en de constructie is vaak een heel werk. Soms is het mogelijk te bewijzen dat een (vermeerderde) grammatica $LR(1)$ is door een sterkere eigenschap te bewijzen, namelijk de *eenvoudige* $LR(1)$-voorwaarde. Deze wordt voor een produktieregel $A \rightarrow \beta$ als volgt gedefinieerd

$$\text{SLR(1)CONTEXT}(A \rightarrow \beta) = \text{LRCONTEXT}(A \rightarrow \beta) \cdot \text{VOLGENDE}(A)$$

Per definitie is dus de LR(1)CONTEXT-verzameling van een produktieregel een deelverzameling van de SLR(1)CONTEXT-verzameling.

Een (vermeerderde) grammatica $G = (N, T, P, S)$ is nu een *eenvoudige* $LR(1)$*-grammatica* (vaak afgekort tot $SLR(1)$*-grammatica*) dan en slechts dan als

uit $\alpha \in$ SLR(1)CONTEXT($A \to \beta$) en $\alpha x \in$ SLR(1)CONTEXT($B \to \beta'$) met $A \to \beta$ en $B \to \beta' \in P, \alpha \in (N \cup T)^*$ en $x \in T^*$ volgt $x = \varepsilon$, $A = B$ en $\beta = \beta'$.

Uit deze definities volgt onmiddellijk:

Stelling 8.4

Iedere $SLR(1)$-grammatica is $LR(1)$.

Het omgekeerde is echter niet waar (zie oefening 8.5). De grammatica G_4 is SLR(1). Voor deze grammatica kunnen we gemakkelijk de VOLGENDE-verzamelingen bepalen met behulp van de in hoofdstuk 7 behandelde technieken.

$$\text{VOLGENDE}(S) = 0$$
$$\text{VOLGENDE}(E) = \{+,), \$\}$$
$$\text{VOLGENDE}(T) = \{\times, +,), \$\}$$
$$\text{VOLGENDE}(F) = \{\times, +,), \$\}$$

Als we de karakteristieke automaat in figuur 8.4 bekijken dan is duidelijk dat de SLR(1)CONTEXT-verzamelingen voldoen aan alle voorwaarden voor een SLR(1)-grammatica. Op grond hiervan is het nu mogelijk een LR-ontleder voor G_4 te ontwerpen. We weten dan een lookahead van een symbool voldoende is om eventuele conflictsituaties op te lossen. In tabel 8.6 wordt dit geillustreerd met een ontleding van de string $a + a \times a + a\$$.

Met behulp van een LR-ontleder kunnen we vaststellen in welke volgorde de reducties moeten worden uitgevoerd en zo wordt dus impliciet de afleidingsboom gegenereerd. Als we ook semantische acties willen inbouwen in de ontleder, dan is dat in het algemeen niet zo moeilijk. Stel dat we semantische bomen willen genereren bij de rekenkundige expressies die door G_4 worden voortgebracht. Dat kan bijvoorbeeld door aan elk element X van de huidige prefix van de zinsvorm een boom T_X te verbinden. T_X is de semantische boom die overeen komt met dat deel van de afleidingsboom dat X als wortel heeft. Steeds als we een symbool $a \in T$ verschuiven, voegen we dit toe aan de huidige prefix en kennen we er een boom T_a aan toe die alleen uit de knoop ⓐ bestaat. Steeds als we de reductie $A \to X_1X_2 \ldots X_k$ toepassen, vervangen we het meest rechtse voorkomen van $X_1X_2 \ldots X_k$ in de huidige prefix door A. Verbonden met deze reductie is een *semantische regel* die beschrijft hoe de bomen $T_{X_1}, T_{X_2}, \ldots, T_{X_k}$ worden gecombineerd tot de boom T_A. Een succesvolle ontleding leidt uiteindelijk tot de toestand 'accepteren', met als huidige prefix het symbool S en als semantische boom T_S.

De semantische regels bij de produktieregels van G_4 kunnen gemakkelijk worden geconstrueerd. In tabel 8.7 zijn zij weergegeven.

Stackinhoud	Stap
1	(ver)schuif a
1, 4	reduceer met $F \rightarrow a$
1, 3	reduceer met $T \rightarrow F$
1, 6	(gebruik één-symbool lookahead) reduceer met $E \rightarrow T$
1, 2	schuif +
1, 2, 8	schuif a
1, 2, 8, 4	reduceer met $F \rightarrow a$
1, 2, 8, 3	reduceer met $T \rightarrow F$
1, 2, 8, 11	(gebruik één-symbool lookahead) schuif ×
1, 2, 8, 11, 10	schuif a
1, 2, 8, 11, 10, 4	reduceer met $F \rightarrow a$
1, 2, 8, 11, 10, 13	reduceer met $T \rightarrow T \times F$
1, 2, 8, 11	(gebruik één-symbool lookahead) reduceer met $E \rightarrow E + T$
1, 2	schuif +
1, 2, 8	schuif a
1, 2, 8, 4	reduceer met $F \rightarrow a$
1, 2, 8, 3	reduceer met $T \rightarrow F$
1, 2, 8, 11	(gebruik één-symbool lookahead) reduceer met $E \rightarrow E + T$
1, 2	schuif $\$$
1, 2, 7	accepteer.

Tabel 8.6

8.4 Theoretische overwegingen

De LR-ontleedtechniek is theoretisch superieur aan de LL-techniek omdat werd aangetoond dat iedere (vermeerderde) $LL(k)$-grammatica ook $LR(k)$ is. Er bestaan echter grammatica's die $LL(1)$ zijn, maar niet $SLR(1)$, (zie oefening 8.6). Iedere $LR(k)$-grammatica definieert een deterministische taal en omgekeerd kan iedere deterministische taal worden gedefinieerd door een $LR(1)$-grammatica. Een taal wordt dus dan en slechts dan gegenereerd door een $LR(k)$-grammatica als de taal door een $LR(1)$-grammatica wordt gegenereerd. $LR(0)$-grammatica's definiëren juist die deterministische talen die voldoen aan de prefix-eigenschap: als x in L voorkomt dan komt geen echte prefix van x in L voor. Als we een einde-inputteken $\$$ gebruiken dan voldoet iedere determinsitische taal $L\$$ aan de prefix-eigenschap. De taal kan dan dus worden gegenereerd door een $LR(0)$-grammatica. Voor praktische toepassingen heeft een compilerbouwer voldoende aan een $LR(1)$-grammatica of een eenvoudige variant daarop, zoals een $SLR(1)$-grammatica. Zie het boek *Formal Languages and their Relation to Automata* door J.E. Hopcroft en J.D. Ullman

Reductie	Semantische regel
$S \to E\$$	$T_S := T_E$
$E \to E + T$	$T_E :=$ construeerboom$(+, T_E, T_T)$
$E \to T$	$T_E := T_T$
$T \to T \times F$	$T_T :=$ construeerboom$(\times, T_T, T_F)$
$T \to F$	$T_T := T_F$
$F \to a$	$T_F := T_a$
$F \to (E)$	$T_F := T_E$

Tabel 8.7

(Addison-Wesley) en *Syntax of Programming Languages: Theory and Practice* door R.C. Backhouse (Prentice-Hall) voor verdere theoretische uitwerking en formele bewijzen van de hier behandelde stof. Een praktische behandeling van de LR-ontleedtechniek is te vinden in *Principles of Computer Design* door A.V. Aho en J.D. Ullman (Addison-Wesley). In dit boek wordt een variant op de LR-grammatica besproken die bekend staat als de $LALR$-grammatica.

8.5 Oefeningen

1. Gegeven zijn een willekeurige contextvrije grammatica $G = (N, T, P, S)$ en twee relaties F en L op $N \cup T$ die als volgt zijn gedefinieerd.

   XFY desd als er een produktieregel in P voorkomt van de vorm
   $X \to Y\gamma, \gamma \in (N \cup T)^*$ en
   XLY desd als er een produktieregel in P voorkomt van de vorm
   $X \to \gamma Y, \gamma \in (N \cup T)^*$.

   Als R^+, R^* en R^{-1} respectievelijk de transitieve insluiting, de reflexieve insluiting en de inverse van een relatie R voorstellen, bewijs dan dat

   $$(\lessdot) = (\doteq)(F^+)$$

   en

   $$(\gtrdot) = (L^+)^{-1}(\doteq)(F^*)$$

   (De relaties $\doteq$, F en L kunnen gemakkelijk worden bepaald. F^+ en L^+ kunnen vervolgens worden bepaald met behulp van het algoritme van Warshall, (zie *JACM* **9**,11-112, januari 1962). $F^* = F^+ \cup I$, waarbij I de identiteitsrelatie voorstelt. Het is nu op grond hiervan mogelijk om de relaties $\lessdot$ en $\gtrdot$ te bepalen.)

2. Bereken de precendentierelaties voor de volgende grammatica

$$S \to A\$$$
$$A \to aABC|CB$$
$$B \to aB|C$$
$$C \to b$$

Is dit een eenvoudige precedentiegrammatica?

3. Toon aan dat de grammatica uit oefening 8.2 $LR(0)$ is, door haar karakteristieke automaat te berekenen.

4. Bewijs dat de grammatica

$$S \to A$$
$$A \to Ab|bBa$$
$$B \to aAc|a|aAb$$

niet $LR(0)$ maar wel $SLR(1)$ is.

5. $G = (N, T, P, S)$ is een willekeurige contextvrije grammatica en k is een positief geheel getal. Veronderstel dat G vermeerderd is zodat er k einde-inputtekens $\$$ staan aan het einde van elke string in $L(G)$. Definieer verder

$$\text{VOLGENDE}_k(A) = \{x \mid \text{lengte}(x) = k \text{ en } S \stackrel{*}{\Rightarrow} y_1 A x y_2 \text{ voor zekere } y_1, y_2 \in T\}.$$

De SLR(k)CONTEXT-verzameling van een produktieregel $A \to \beta$ in G wordt dan gedefinieerd als

$$\text{SLR}(k)\text{CONTEXT}(A \to \beta) = \text{LRCONTEXT}(A \to \beta) \cdot \text{VOLGENDE}_k(A).$$

Gebruik deze definitie om formeel een SLR(k)-grammatica te definiëren en aan te tonen dat

$$S \to A\$$$
$$A \to BaCD$$
$$B \to b$$
$$C \to B|Ea$$
$$D \to ba|Dba$$
$$E \to b$$

$LR(1)$ is maar niet $SLR(k)$ voor enige $k > 0$.

6. Bewijs dat de grammatica

$$S \to A\$$$
$$A \to BaBb|CbCa$$
$$B \to \varepsilon$$
$$C \to \varepsilon$$

$LL(1)$ is maar niet $SLR(1)$.

7. Generaliseer de definitie van een $LR(1)$-grammatica tot die van een $LR(k)$-grammatica.
8. Schrijf een Pascal-programma voor een LR-ontleder voor eenvoudige rekenkundige expressies. Het programma moet toegelaten expressies in de variabelen $\{a, b, c, \ldots, z\}$ accepteren en als resultaat de bijbehorende semantische boom opleveren. Bij incorrecte expressies moet een foutmelding worden gegeven.

Index

Wie een taal gaat bestuderen, of deze nu formeel is of niet, heeft één groot voordeel: iedereen is expert in minstens één taal, zijn of haar moedertaal. Velen van ons kennen tegenwoordig ook een programmeertaal, zoals ALGOL, Pascal, BASIC of FORTRAN. Bij het programmeren blijkt echter dat computers niet zoals mensen aan een half woord genoeg hebben. Syntaxis en semantiek van programmeertalen zijn daarom een belangrijk studieobject en zeker als men zich wil gaan verdiepen in compilerbouw is kennis van de formele-taaltheorie noodzakelijk.

In dit boek worden hoofdzakelijk reguliere en contextvrije grammatica's behandeld, omdat deze bij het definiëren van programmeertalen een belangrijke rol spelen.

> ACADEMIC SERVICE

ISBN 90 6233 242 0
NUGI 852/112